Outline

Chapter 8: A C++ Introduction For Python Programmers
Section 8.1 Introduction
Section 8.2 C++ History and Background

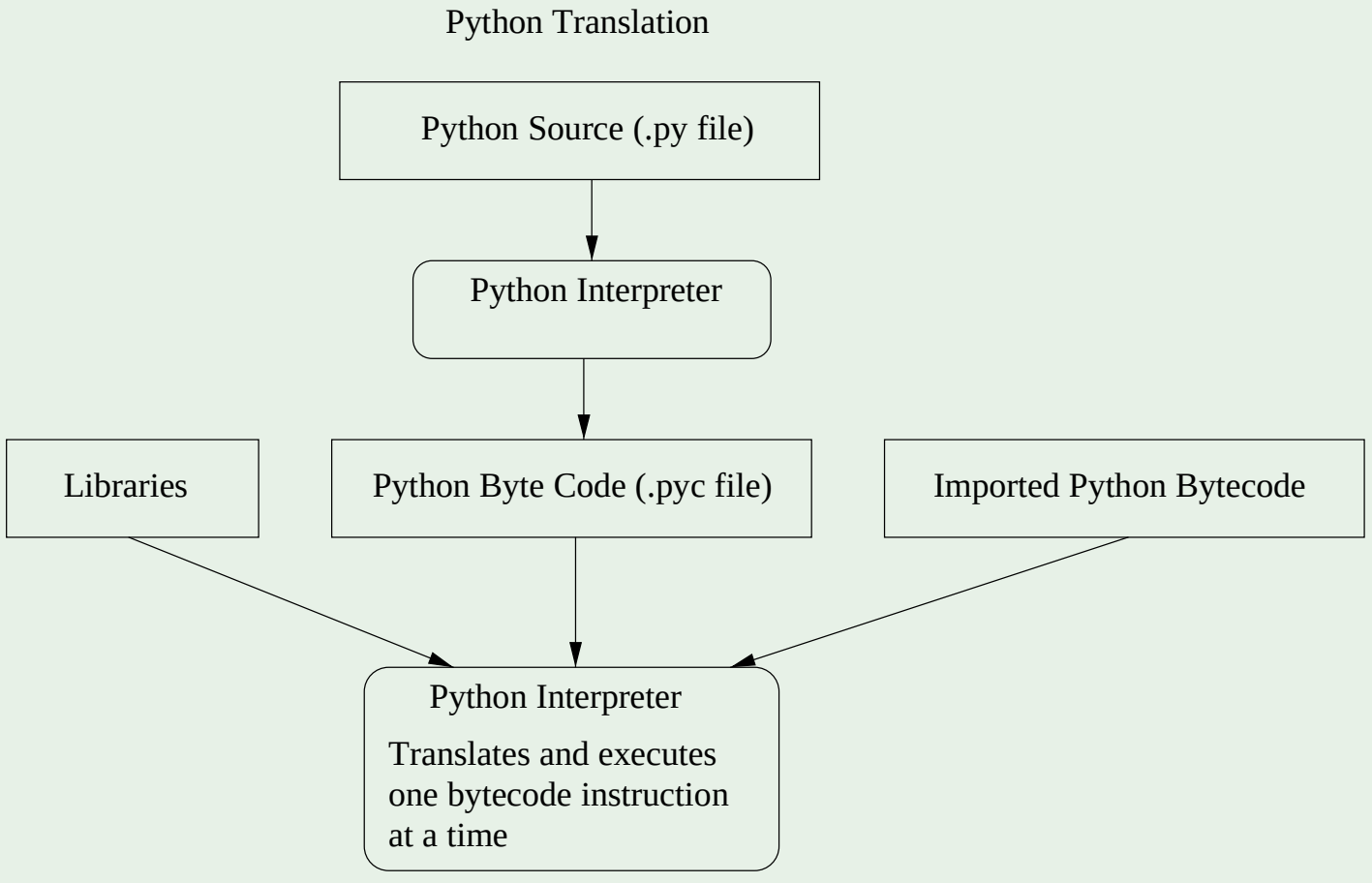
Chapter 10: C++ Dynamic Memory
Section 10.1 Introduction
Section 10.2 C++ pointers

History

Read about the C++ history and background in the textbook and see some other resources as well.

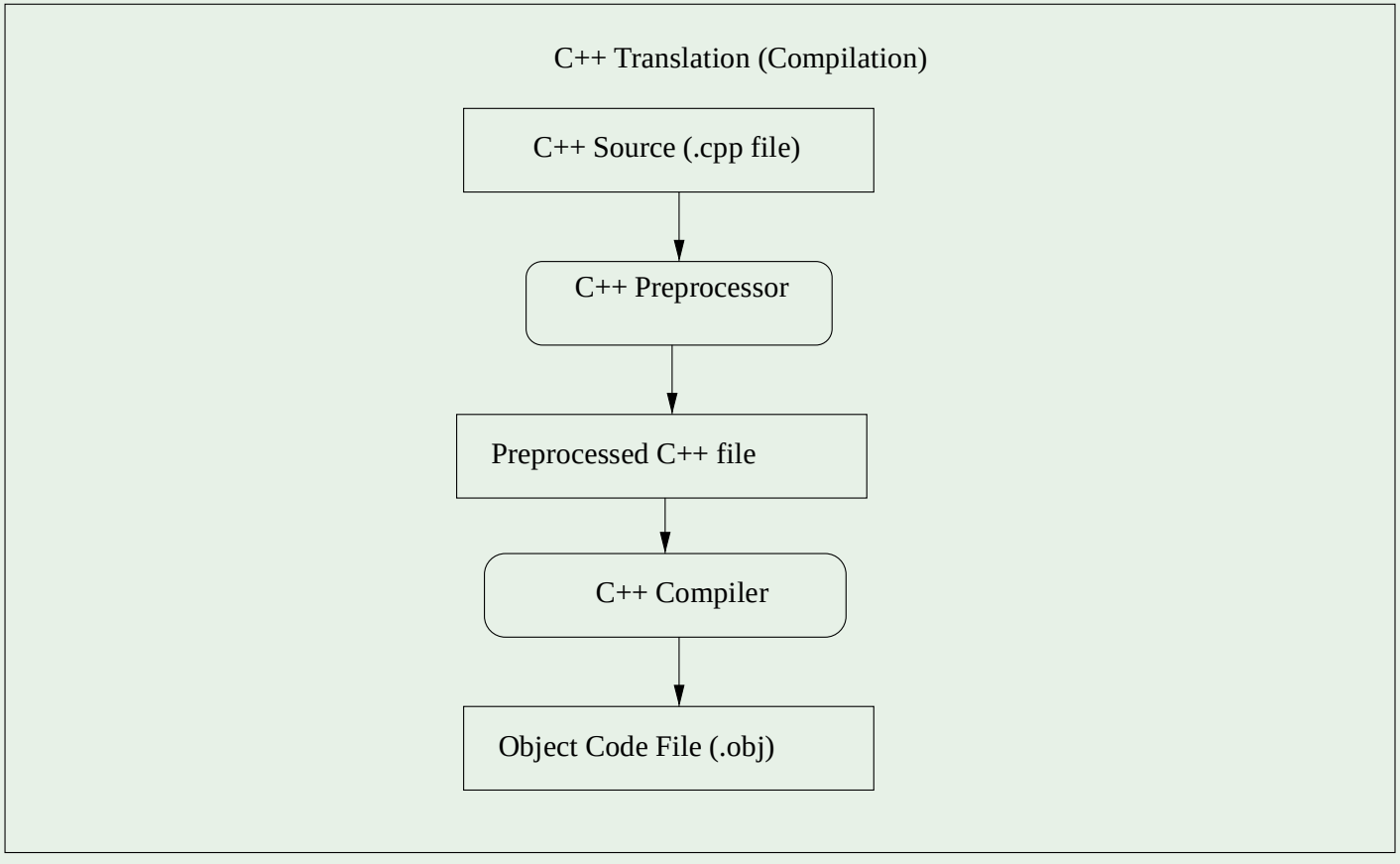
Python Translation Process

Interpretation



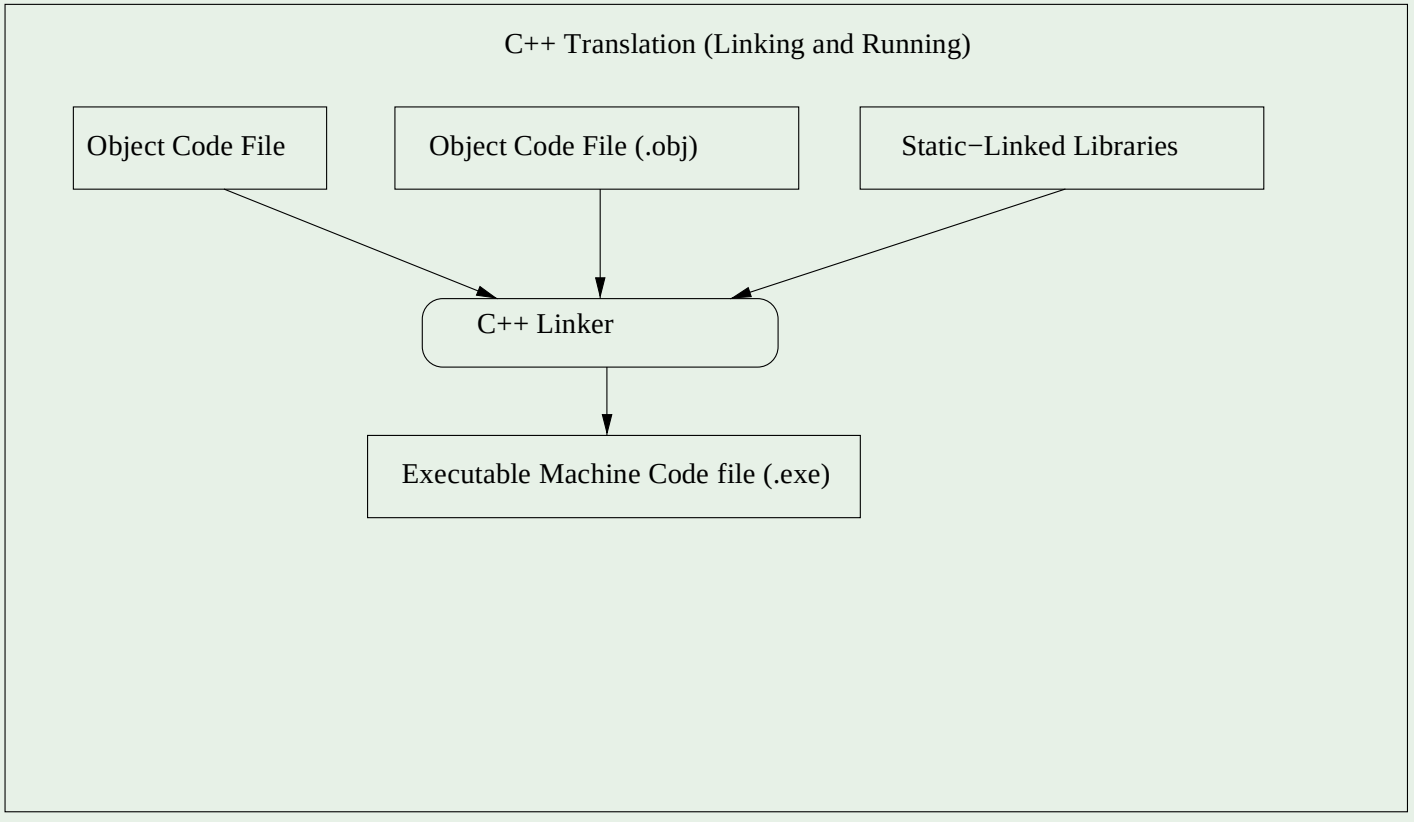
C++ Translation Process - “Building” a Program

Compilation



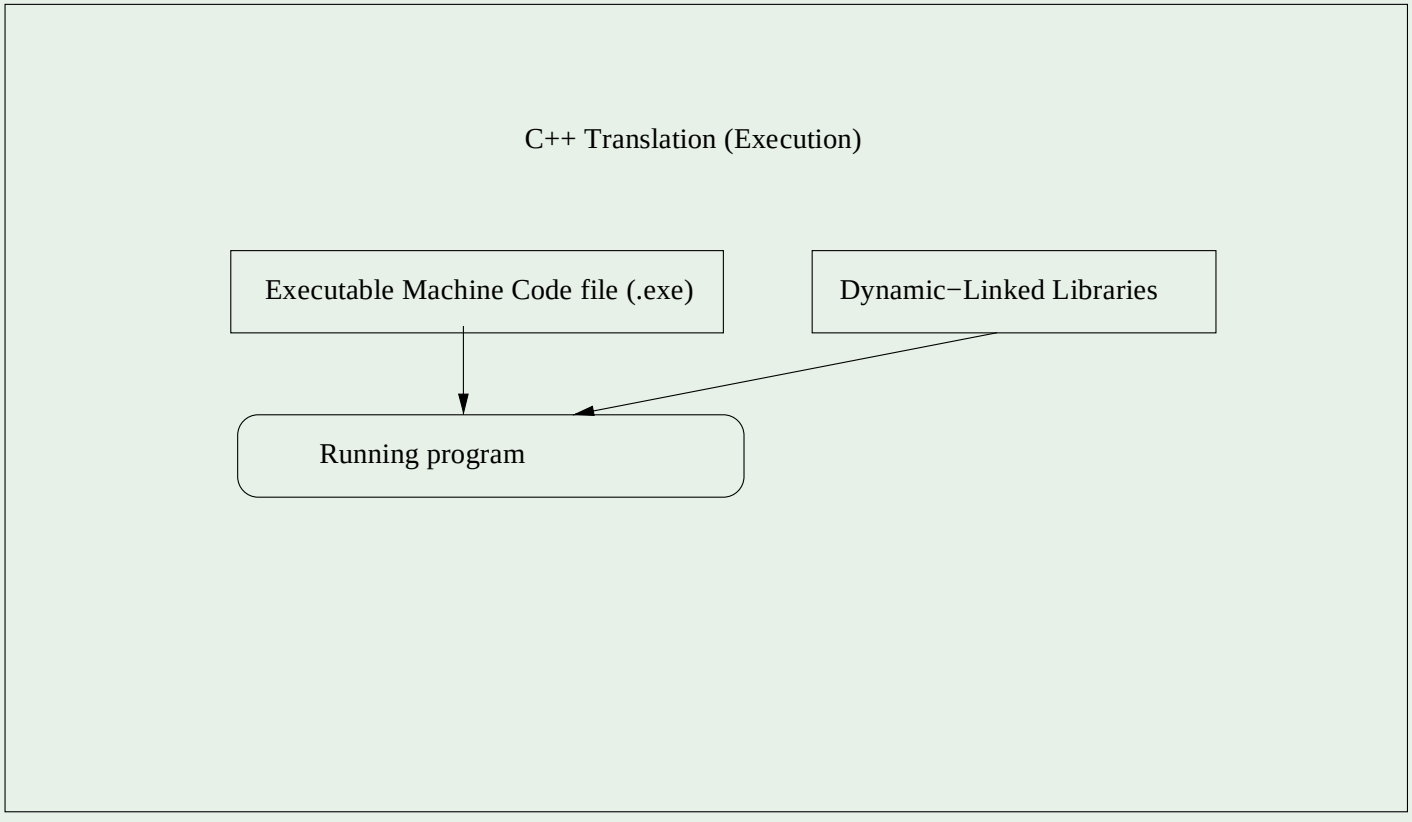
C++ Translation Process - “Building” a Program

Linking



C++ Translation Process - “Building” a Program

Execution



Build

The make Utility

- A script (short program) which tells the computer to perform the steps of building an executable file:
- Preprocessing and compiling a .cpp file into an object(.o) file.
- Linking the object file with other machine code files to produce an executable (.exe) file.
- Only perform operations if the output file is older than the input file.

Build

Eclipse Console View

```
**** Build of configuration Debug for project HelloWorld ****
**** Internal Builder is used for build ****

g++ -O0 -g3 -Wall -c -fmessage-length=0 -ohello.o ..\hello.cpp
g++ -oHelloWorld.exe hello.o
Build complete for project HelloWorld
Time consumed:2266 ms.
```

Storing Variables In Python and C++

Storing Variables In Python:

- values of Python variables (objects) are found by references (addresses) associated with the variable names.
- the memory used to store the value (the object) is allocated when it is needed.
- the object's information includes its type and a reference count.
- assignment in Python changes the reference, not the object itself.

Storing Variables In Python and C++

Storing Variables In C++:

- values of C++ variables are in locations directly associated with the variable names.
- the memory used to store declared variables must be allocated at compile time.
- no reference counting is used; the variable's memory is deallocated when the program leaves its scope.
- assignment in C++ changes the variable's value itself.

Storing Variables In Python and C++

Objects In Python:

- assignment of a variable does not change the object it refers to.

- the memory used to store the value (the object) is created when it is needed.

Storing Variables In Python and C++

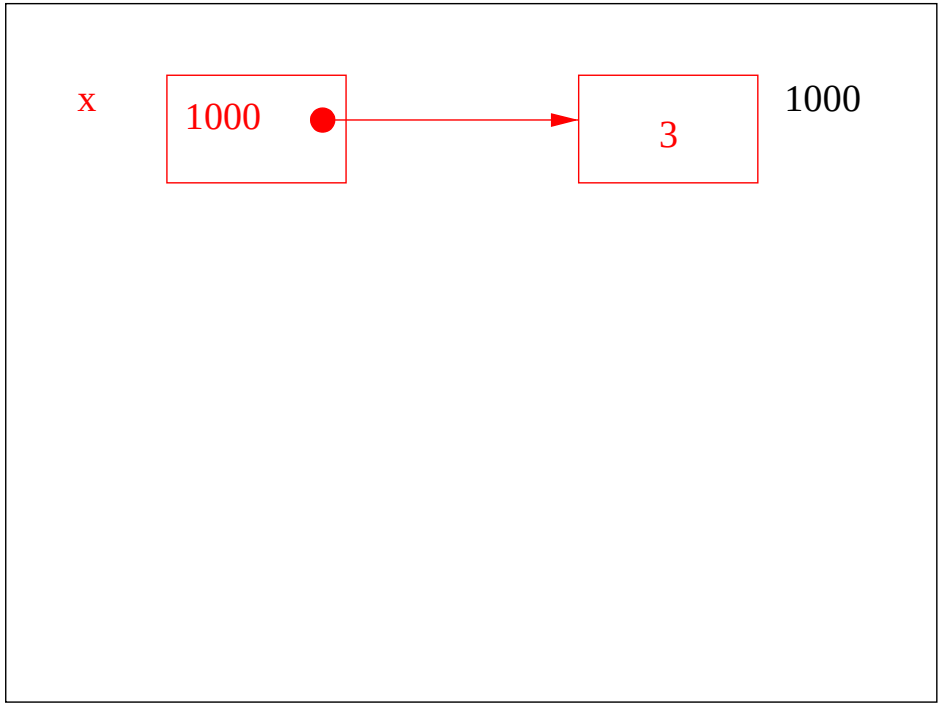
Objects In C++:

- Assignment of a variable changes the attributes of the object in the variable's location.
  (The assignment operator = can be overloaded in C++. The default behavior is to assign the attribute values of the object on the right side to those of the object on the left.

- The memory used to store the value (the object) is created when the program enters its scope.

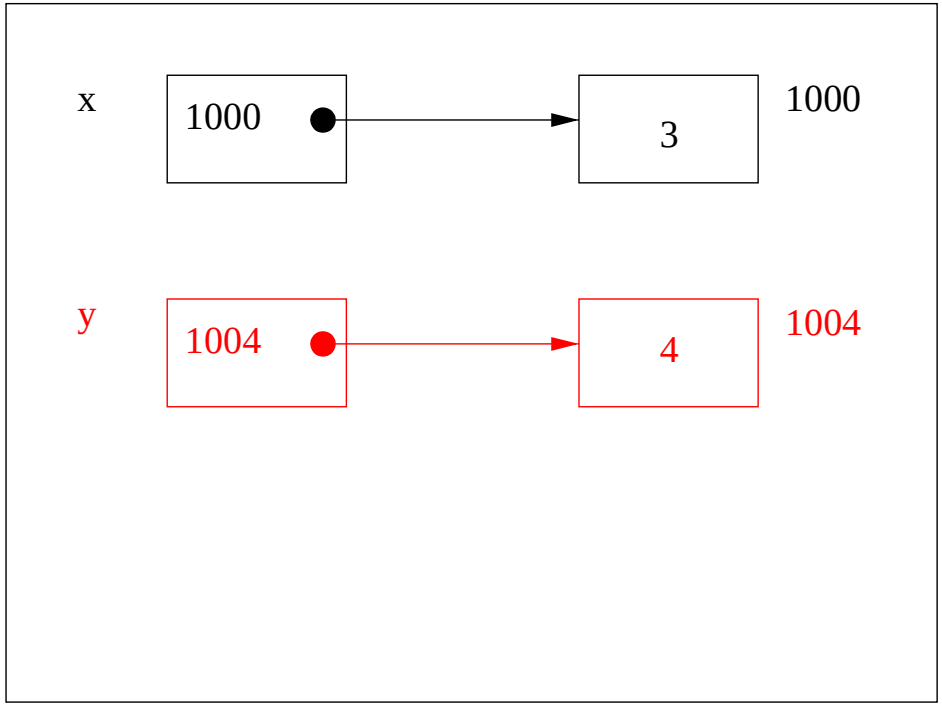
Python memory example

```
x = 3
y = 4
z = x
x = y
```



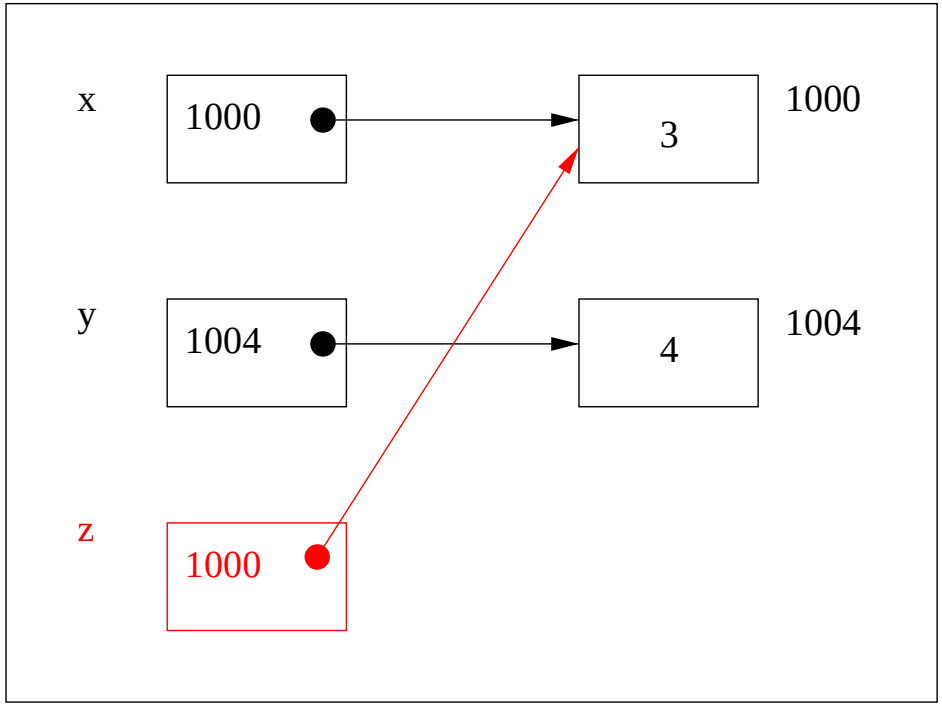
Python memory example

```
x = 3
y = 4
z = x
x = y
```



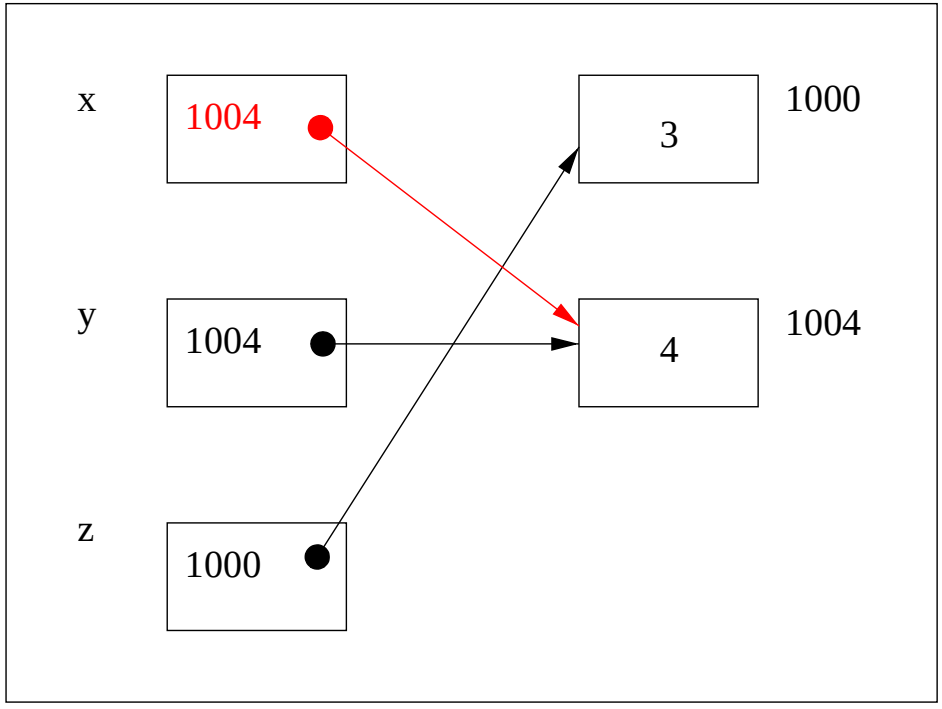
Python memory example

```
x = 3
y = 4
z = x
x = y
```



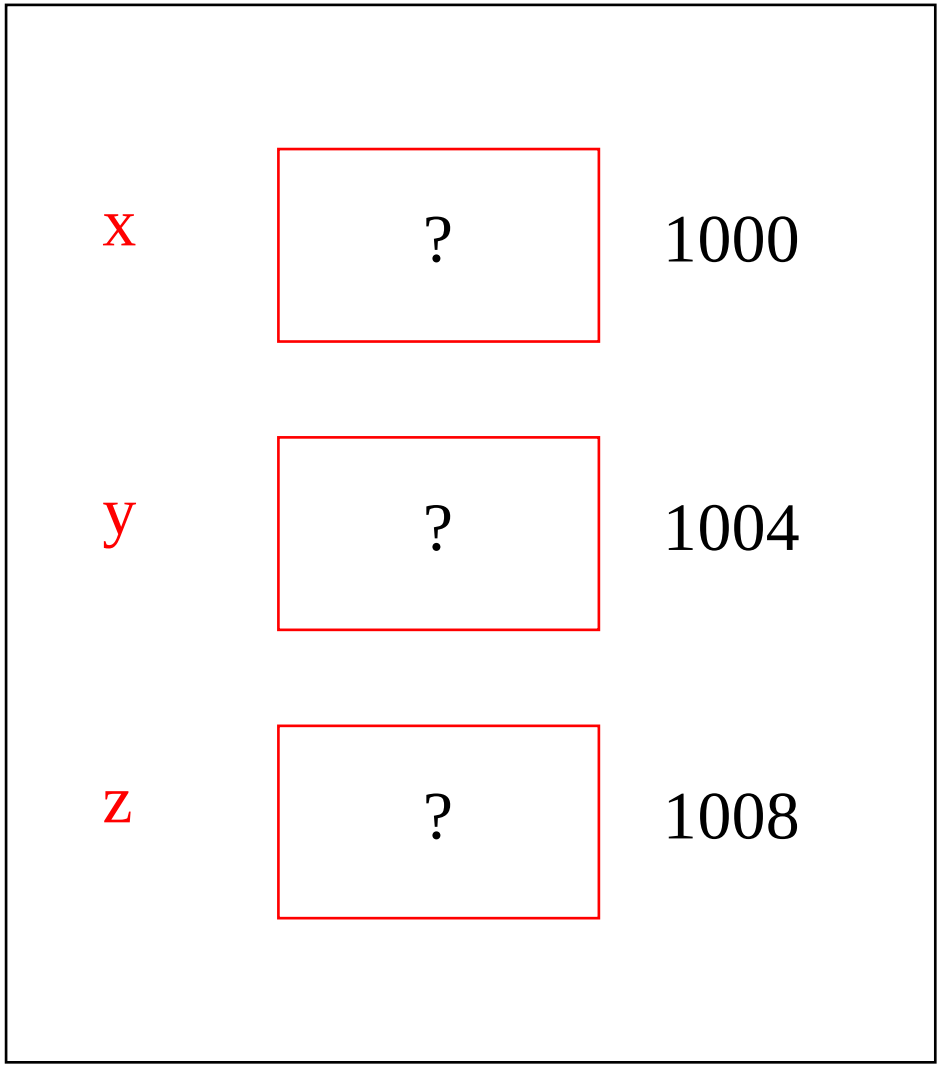
Python memory example

```
x = 3
y = 4
z = x
x = y
```



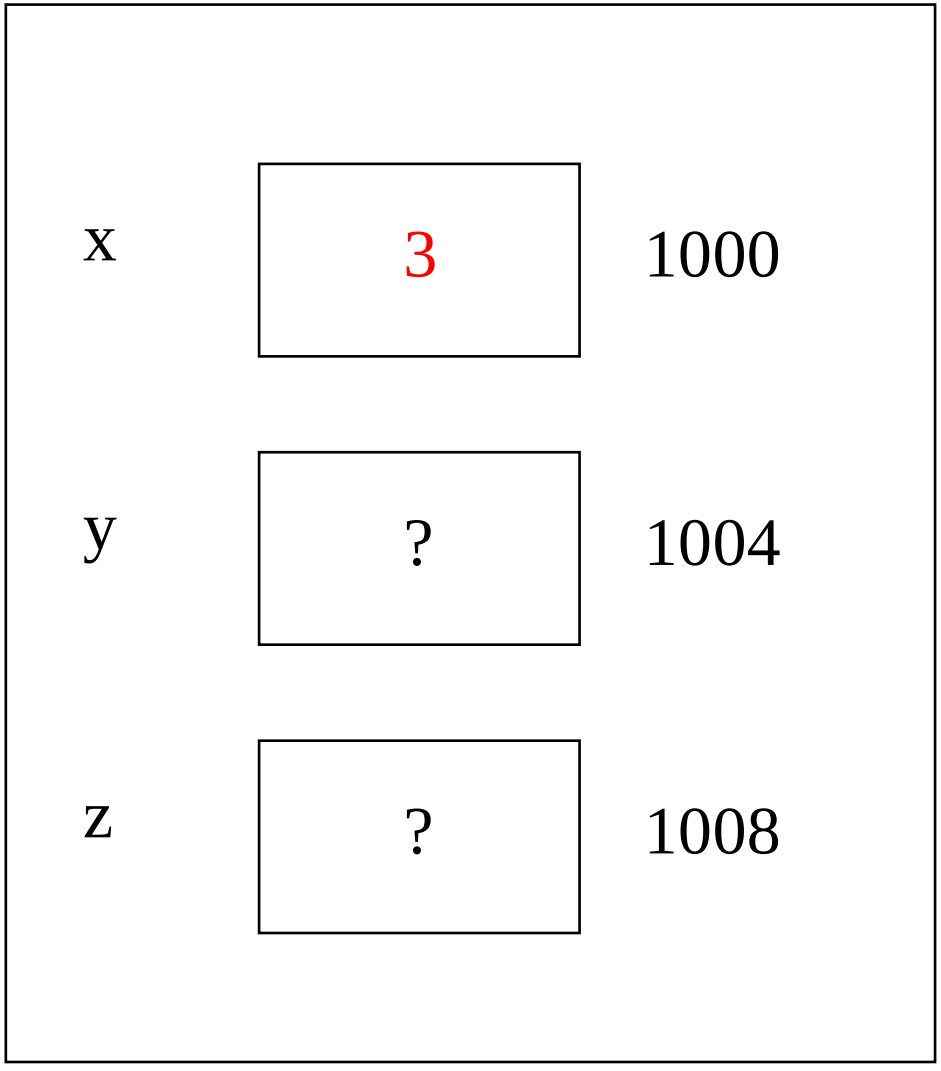
C++ memory example

```
int x, y, z;
x = 3;
y = 4;
z = x;
x = y;
```



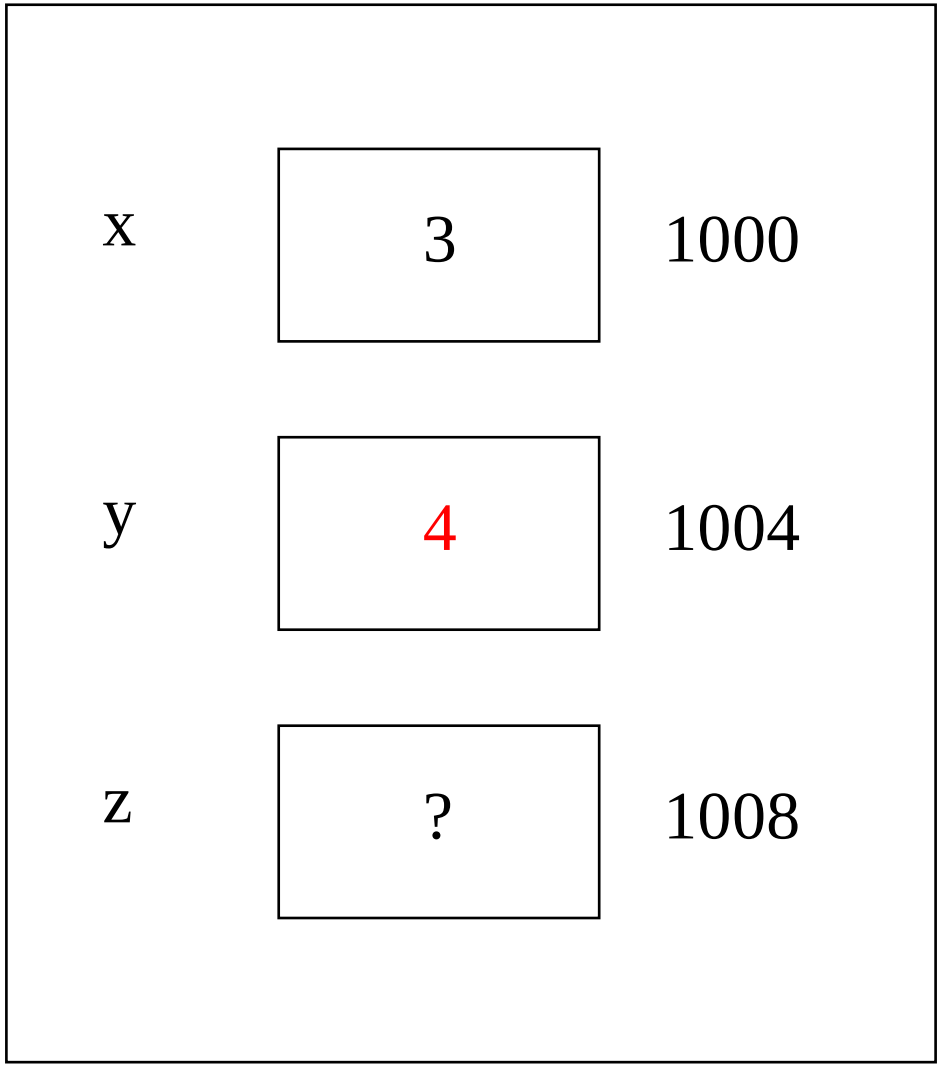
C++ memory example

```
int x, y, z;
x = 3;
y = 4;
z = x;
x = y;
```



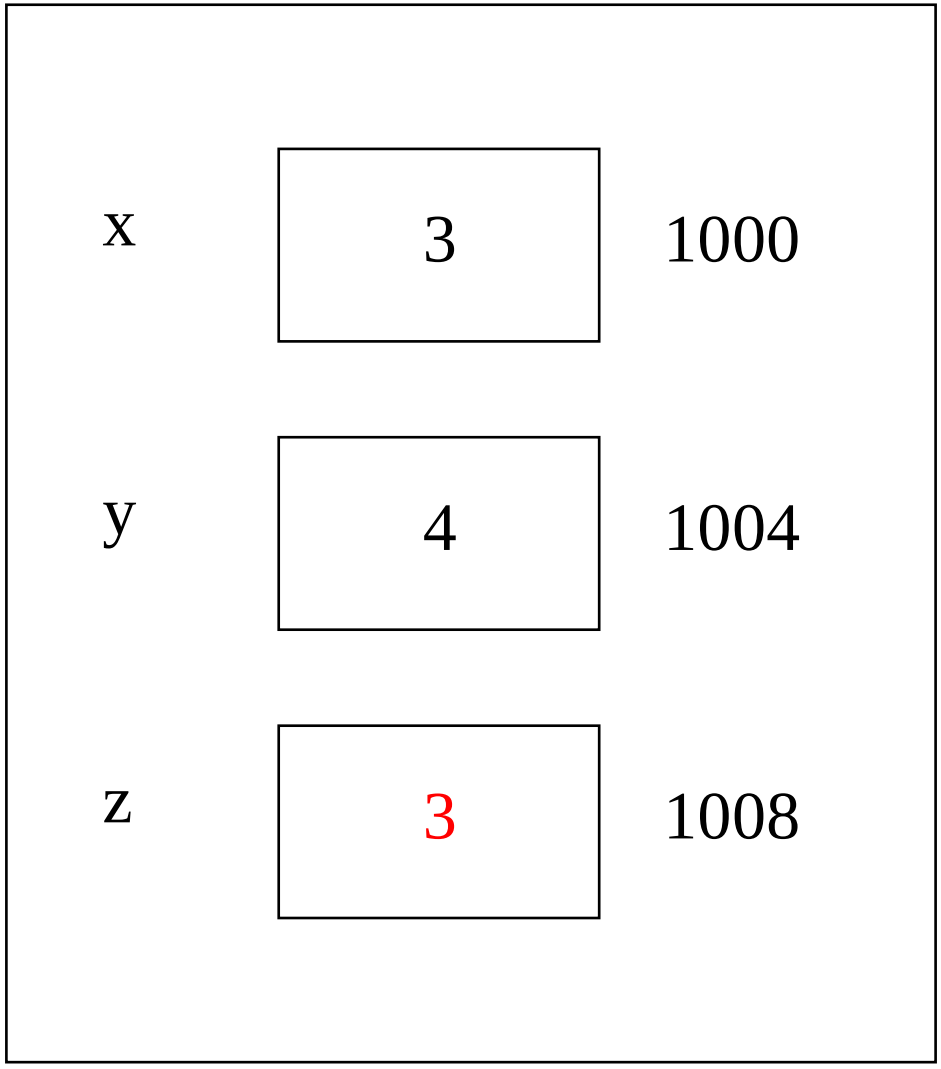
C++ memory example

```
int x, y, z;
x = 3;
y = 4;
z = x;
x = y;
```



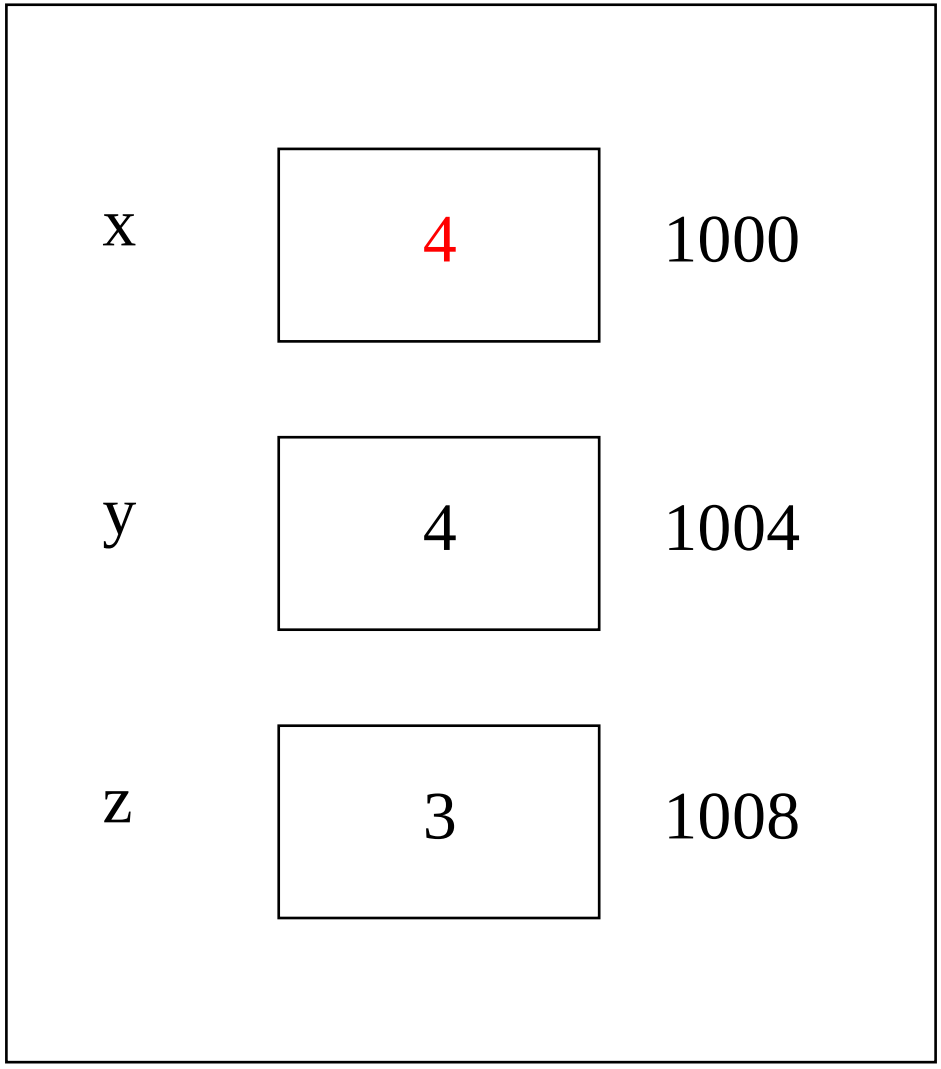
C++ memory example

```
int x, y, z;
x = 3;
y = 4;
z = x;
x = y;
```



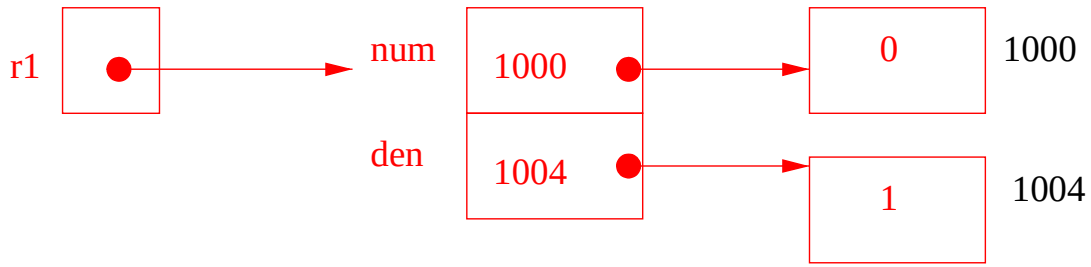
C++ memory example

```
int x, y, z;
x = 3;
y = 4;
z = x;
x = y;
```



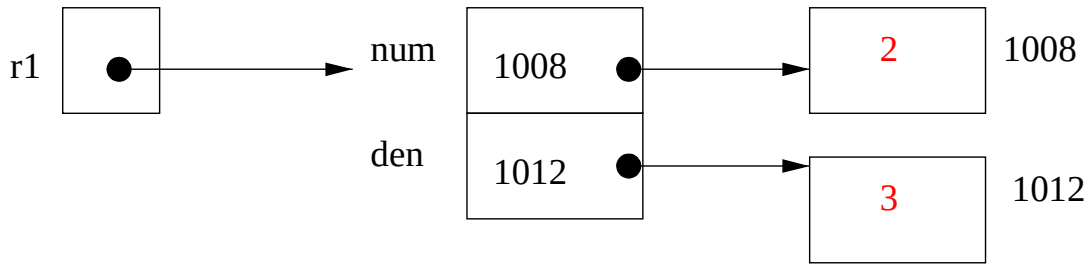
Python object example

```
r1 = Rational()
r1.set(2, 3)
r2 = r1
r1.set(1, 3)
```



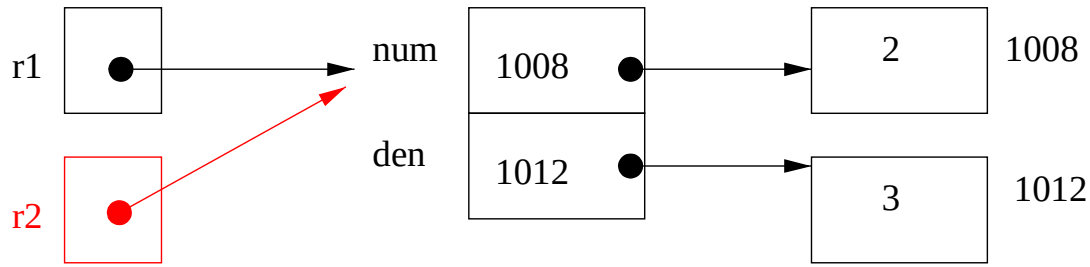
Python object example

```
r1 = Rational()
r1.set(2, 3)
r2 = r1
r1.set(1, 3)
```



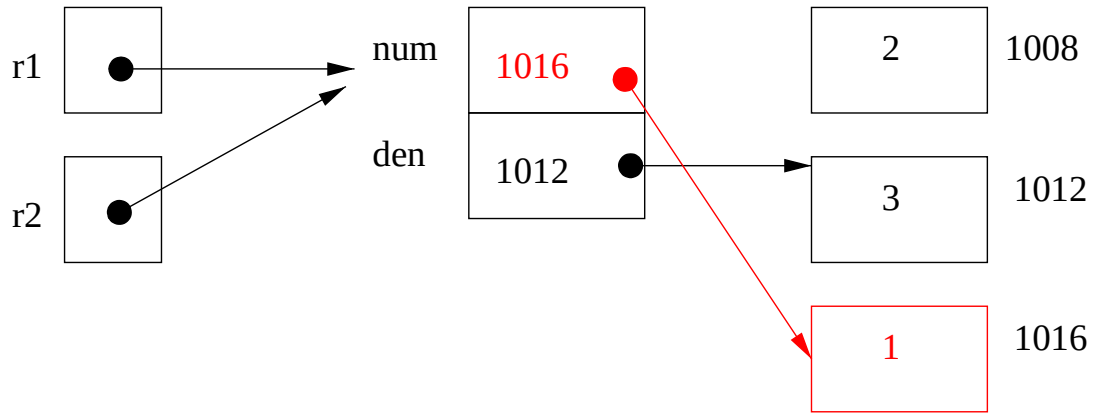
Python object example

```
r1 = Rational()
r1.set(2, 3)
r2 = r1
r1.set(1, 3)
```



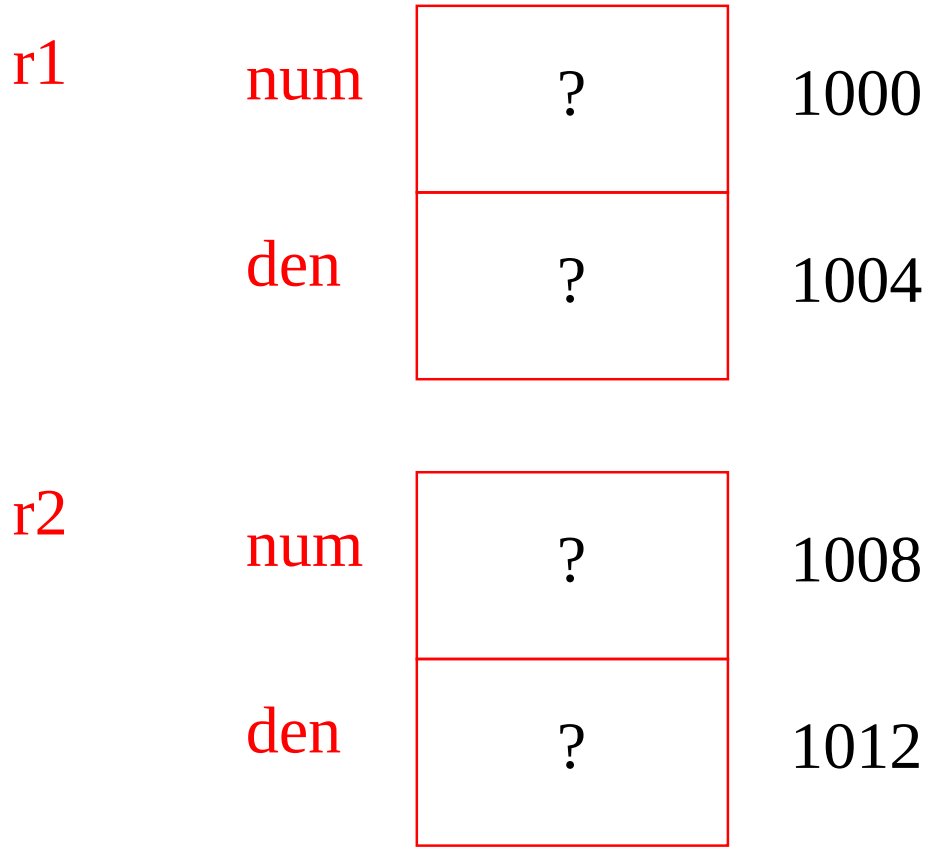
Python object example

```
r1 = Rational()
r1.set(2, 3)
r2 = r1
r1.set(1, 3)
```



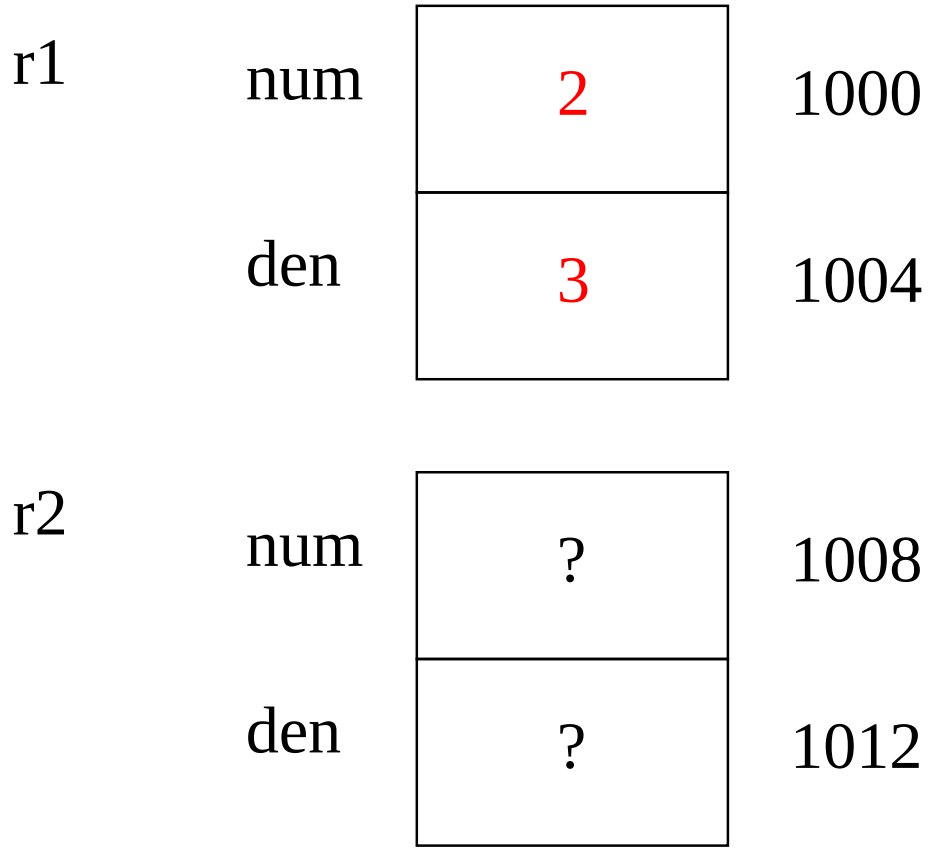
C++ object example

```
Rational r1, r2;
r1.set(2, 3);
r2 = r1;
r1.set(1, 3);
```



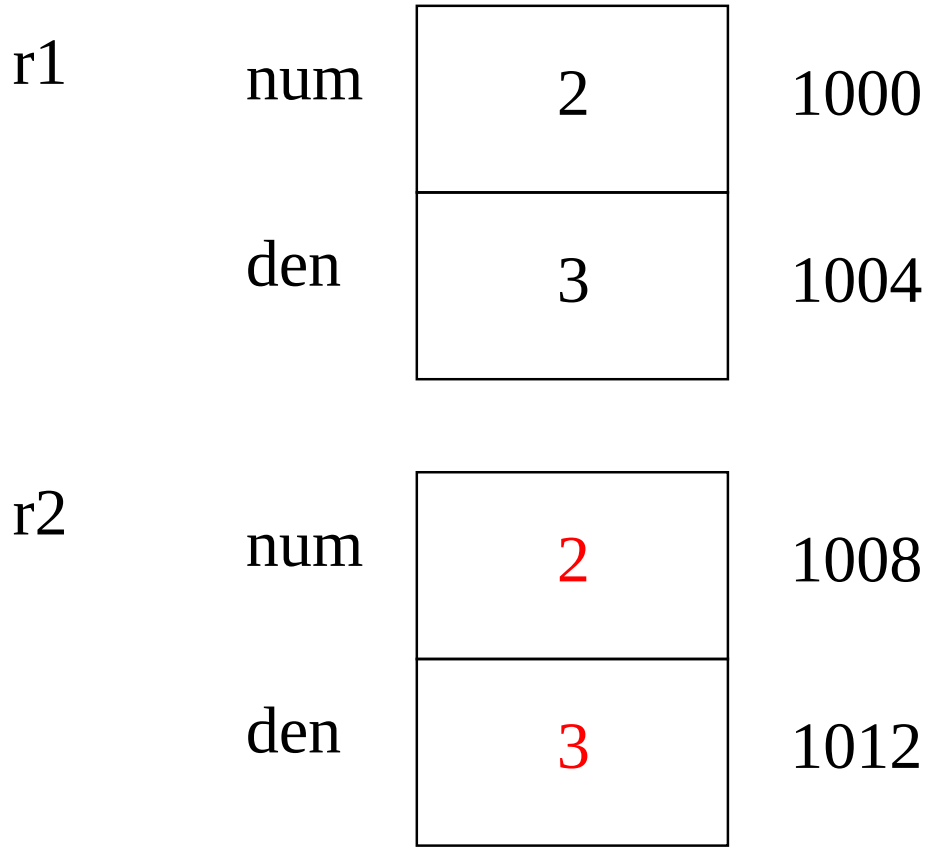
C++ object example

```
Rational r1, r2;
r1.set(2, 3);
r2 = r1;
r1.set(1, 3);
```



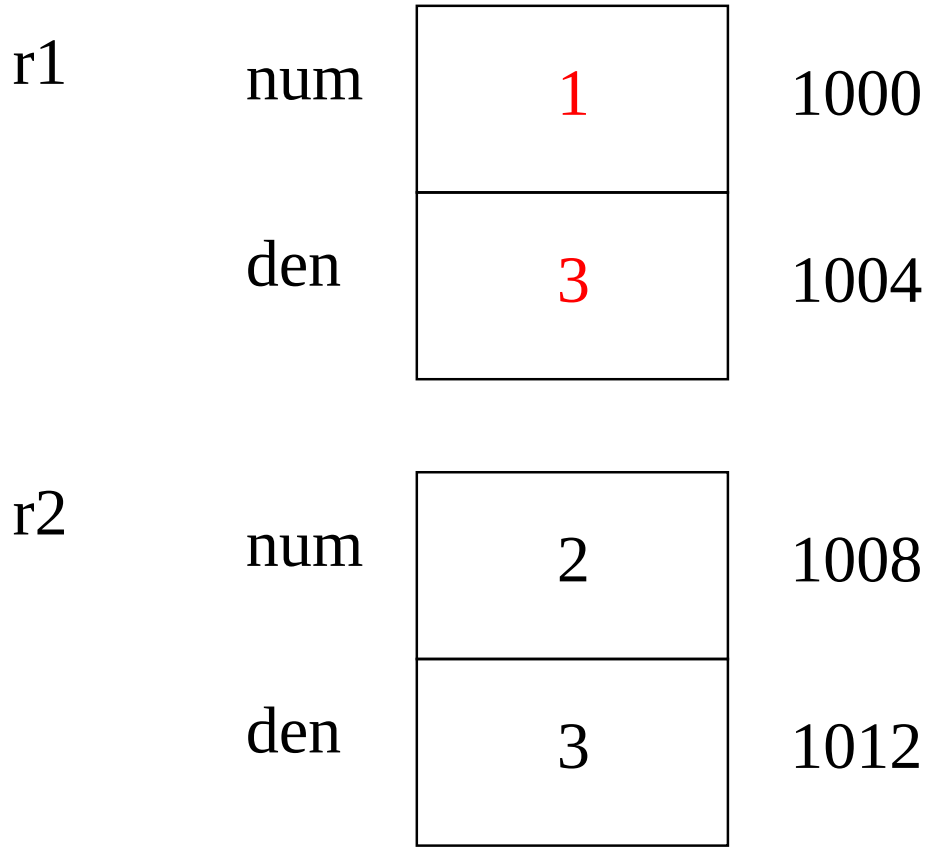
C++ object example

```
Rational r1, r2;
r1.set(2, 3);
r2 = r1;
r1.set(1, 3);
```



C++ object example

```
Rational r1, r2;
r1.set(2, 3);
r2 = r1;
r1.set(1, 3);
```



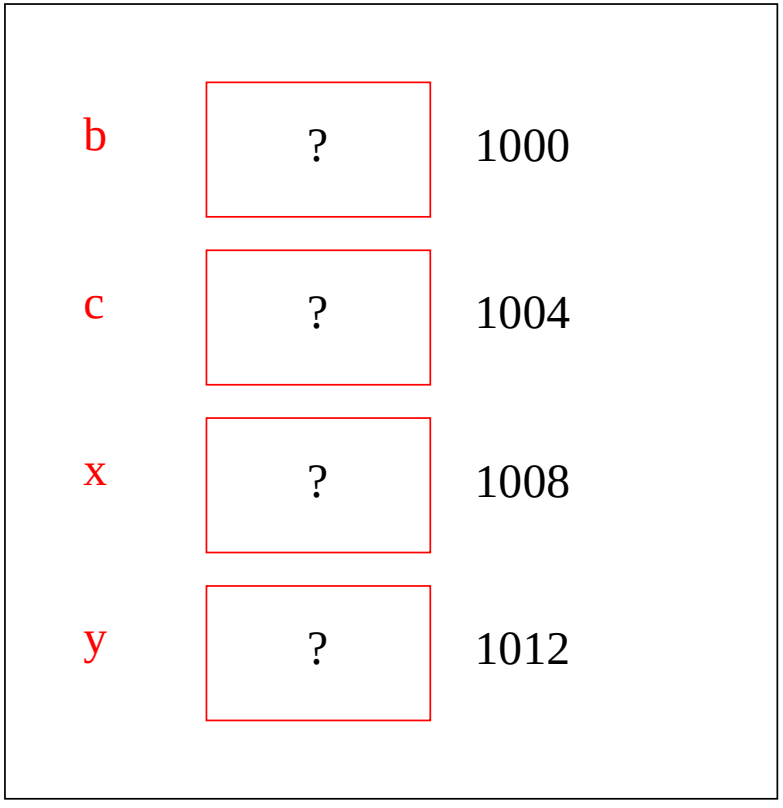
Pointer syntax in C++

Pointer declaration:

- In C++, pointer variable stores a *memory address*.
- The pointer has to be defined with a specific type, which indicates how the data at that address should be interpreted.
- C++ pointers are declared using the asterisk (*) as a prefix to the variable name. We call the unary asterisk operator *dereference operator*.
- So, a pointer declaration reserves space for the address of an object of the type given before the * symbol. (A C++ pointer plays the role of a reference in the Python style.)
- The ampersand (&), or reference operator, is the opposite of *. It gives the **address** of the variable after it.

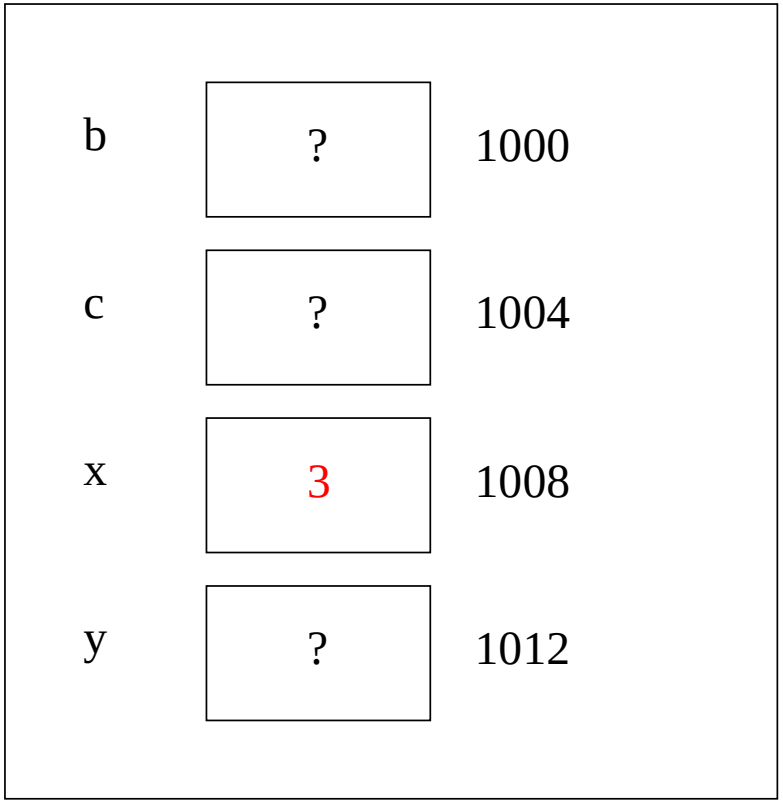
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



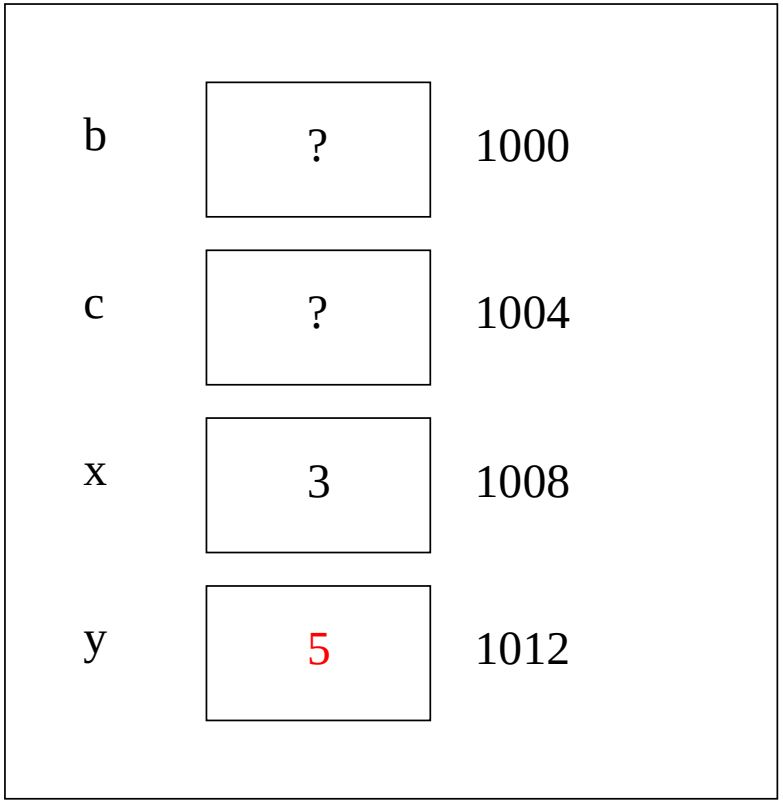
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



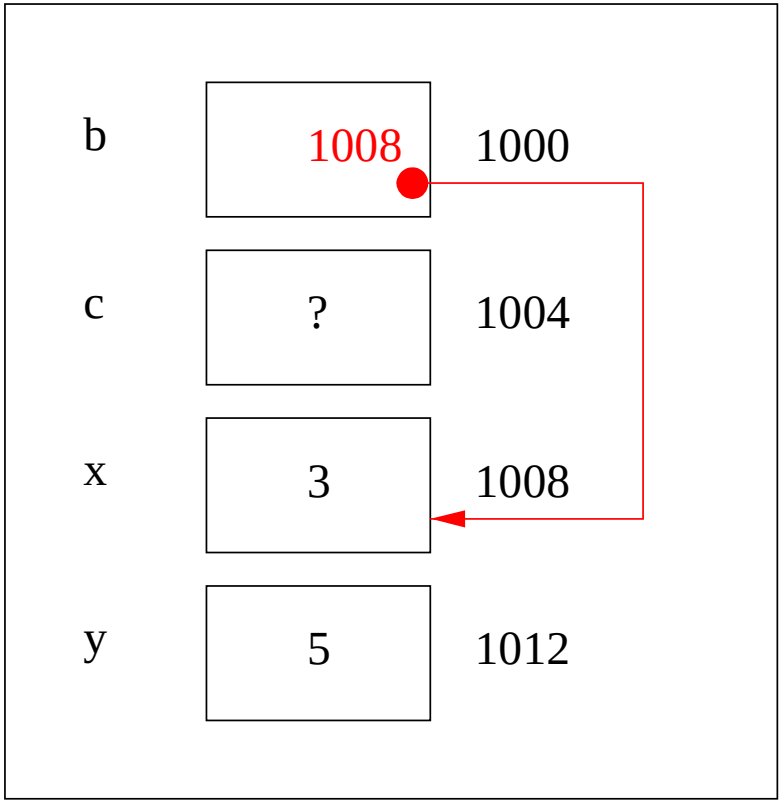
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



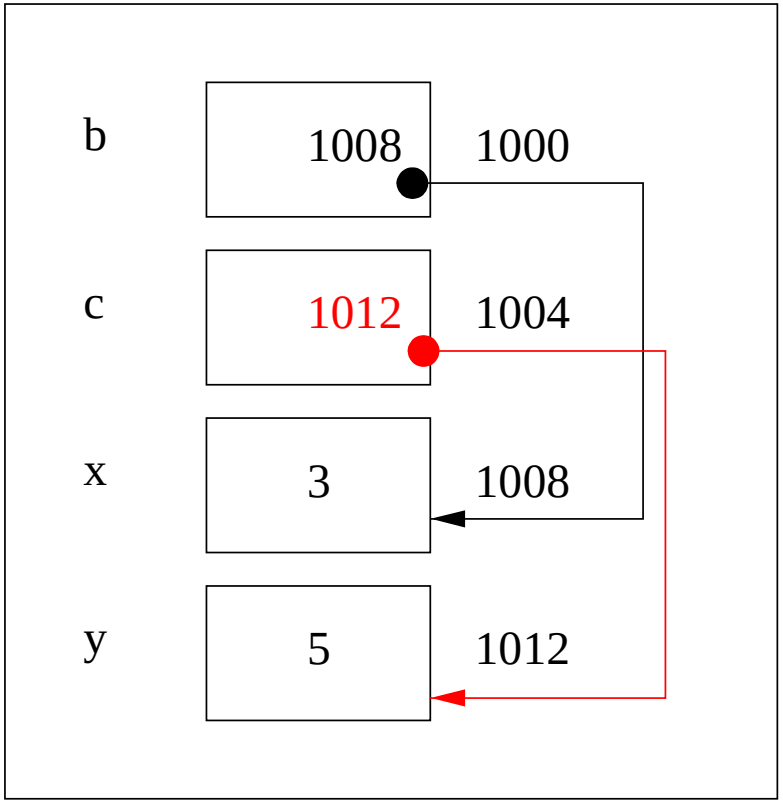
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



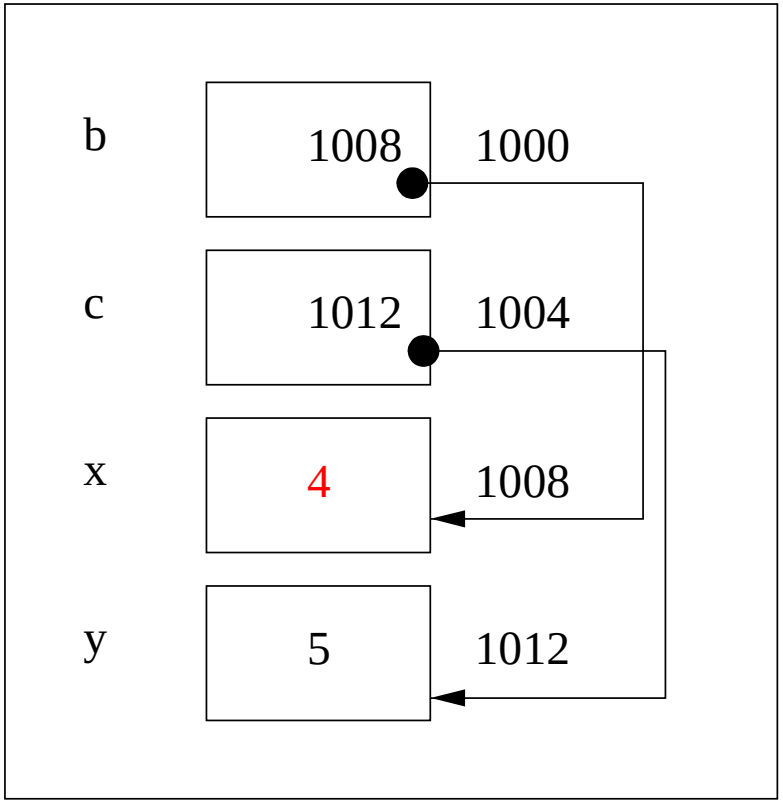
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



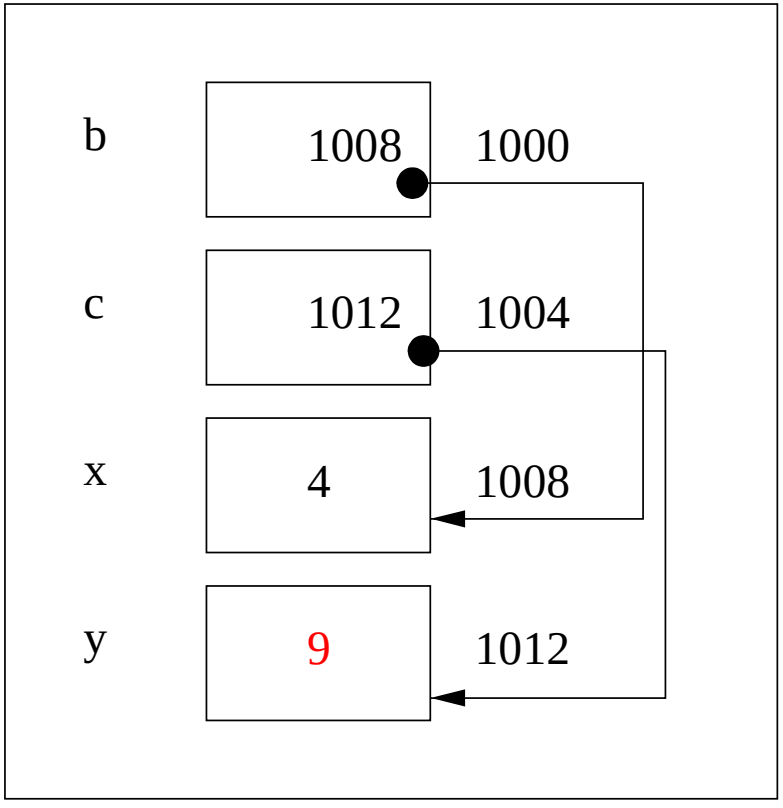
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



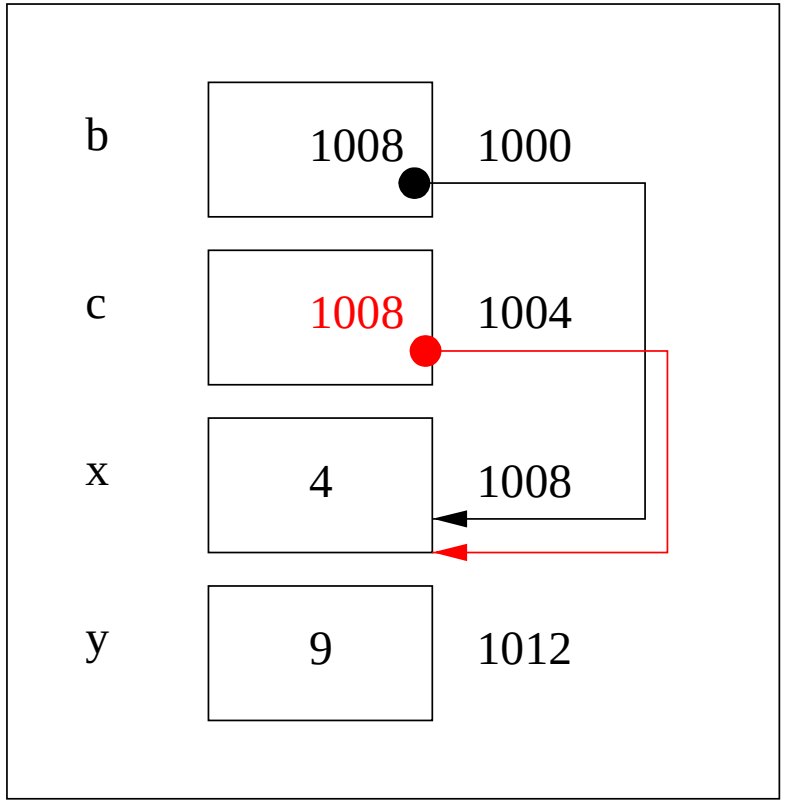
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



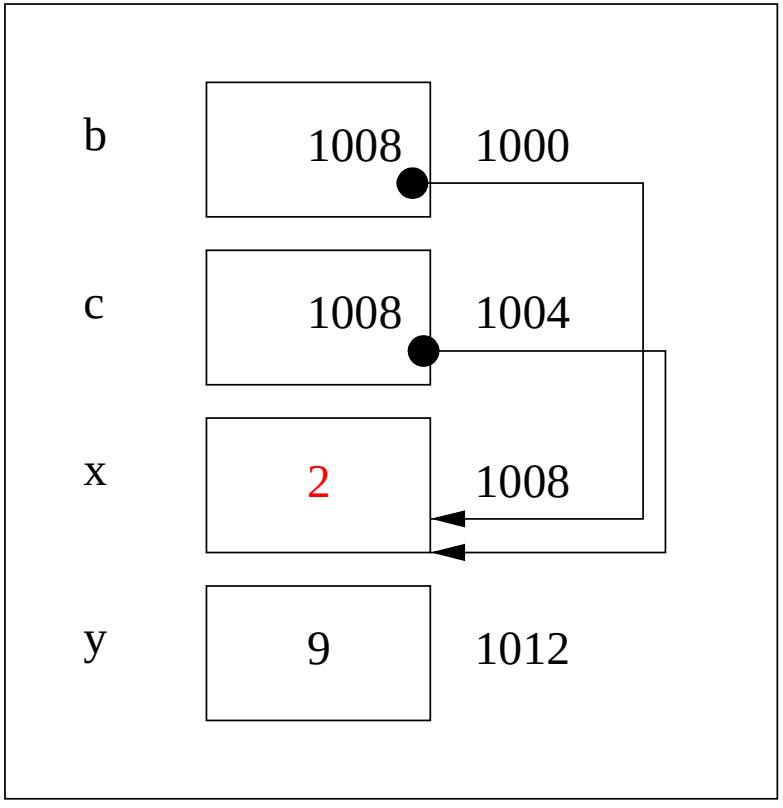
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



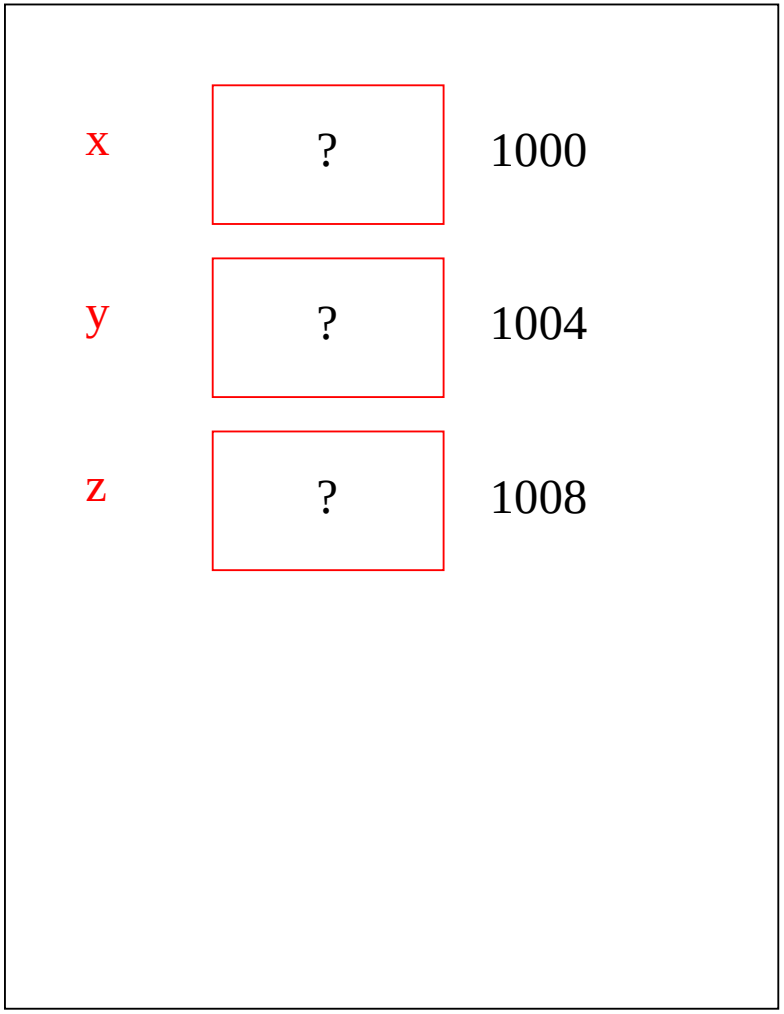
C++ pointer example 1

```
int *b, *c, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```



C++ pointer example 2

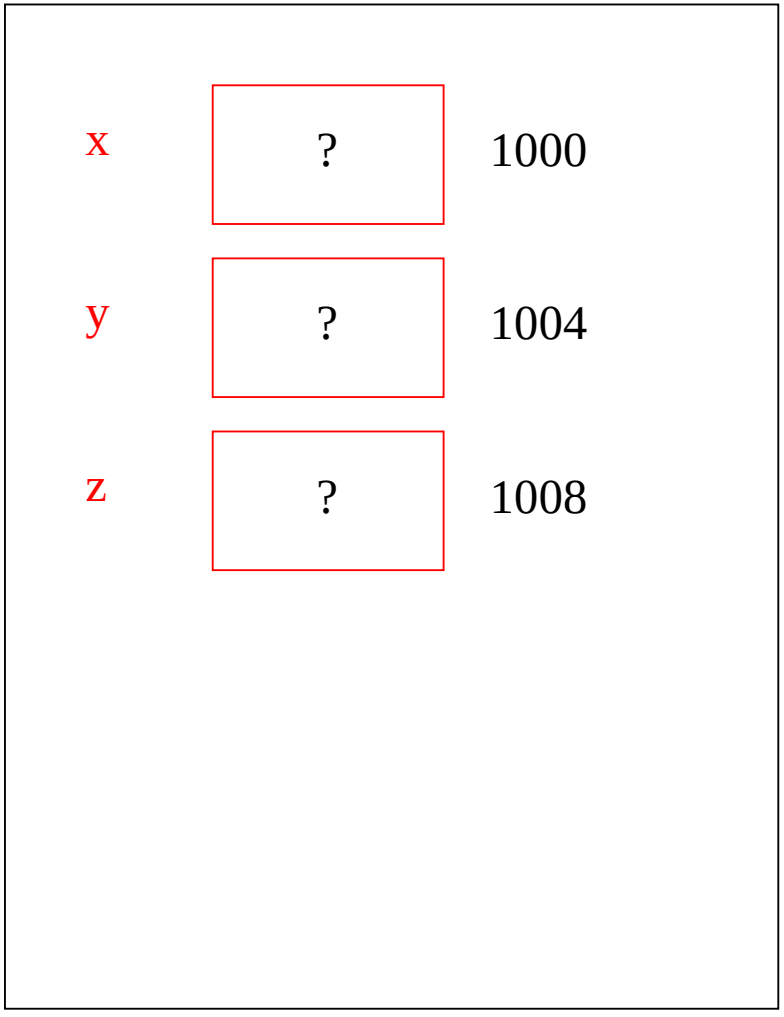
```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



C++ pointer example 2

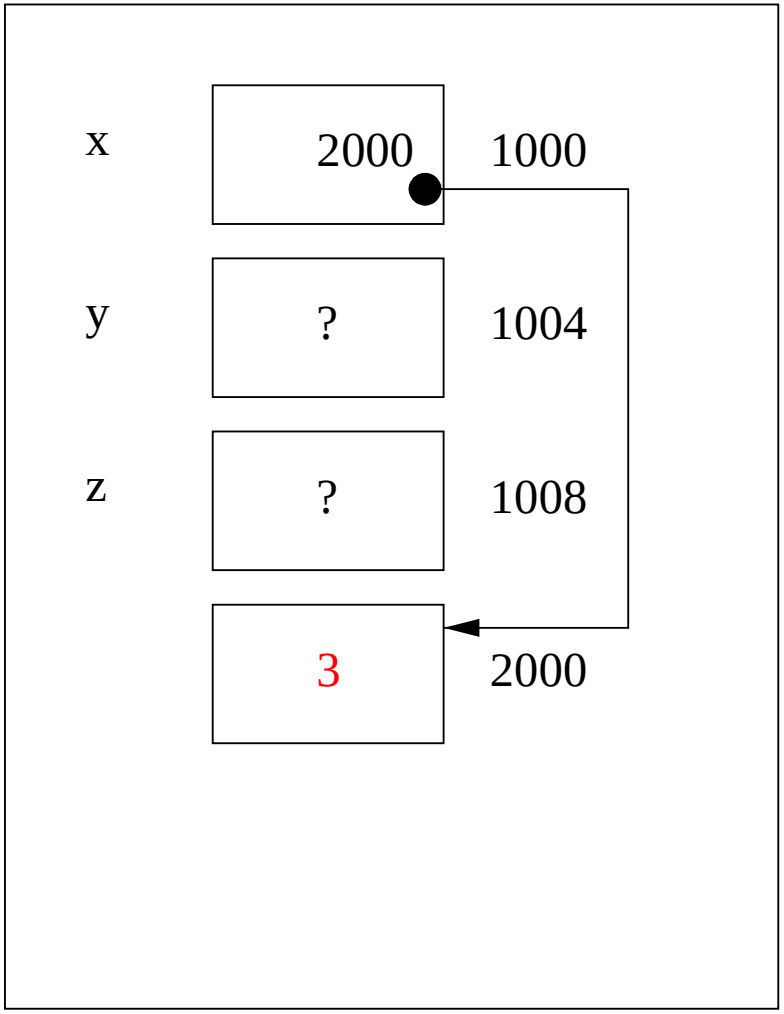
```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```

new statement allocates dynamic memory and returns the starting address



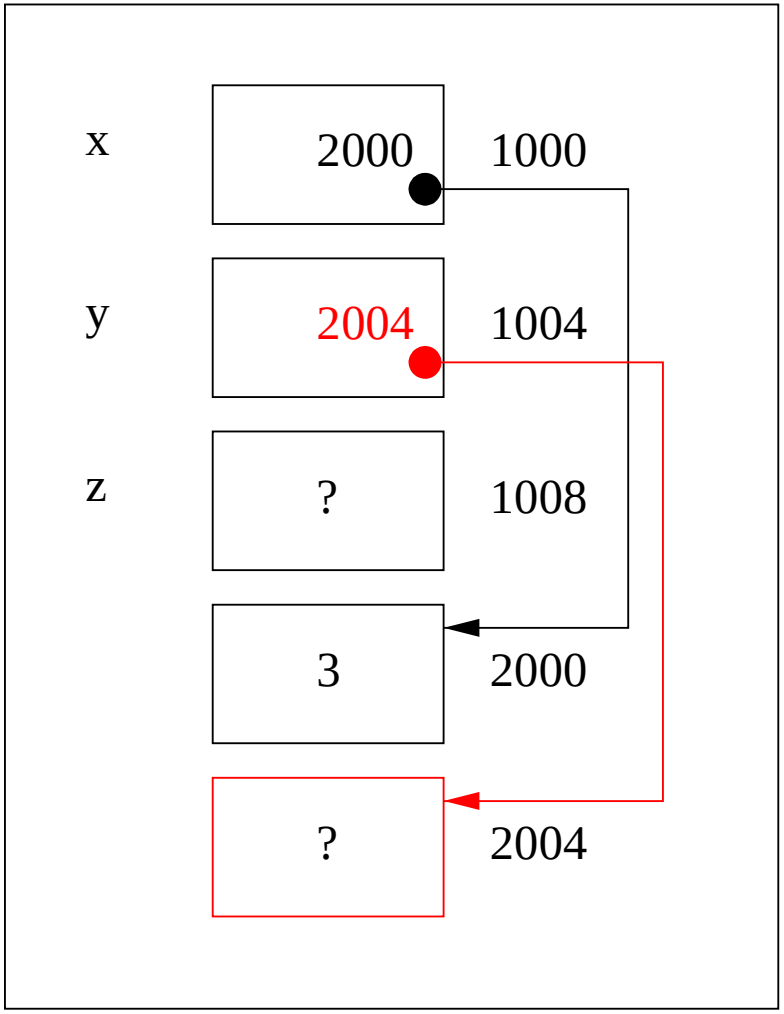
C++ pointer example 2

```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



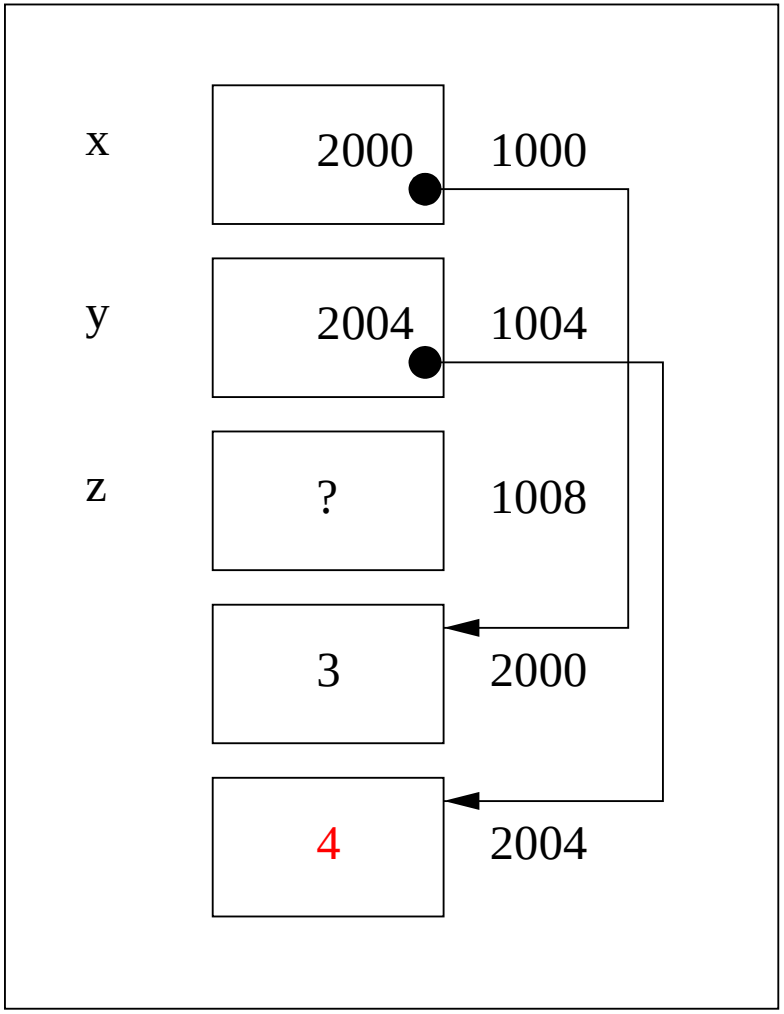
C++ pointer example 2

```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



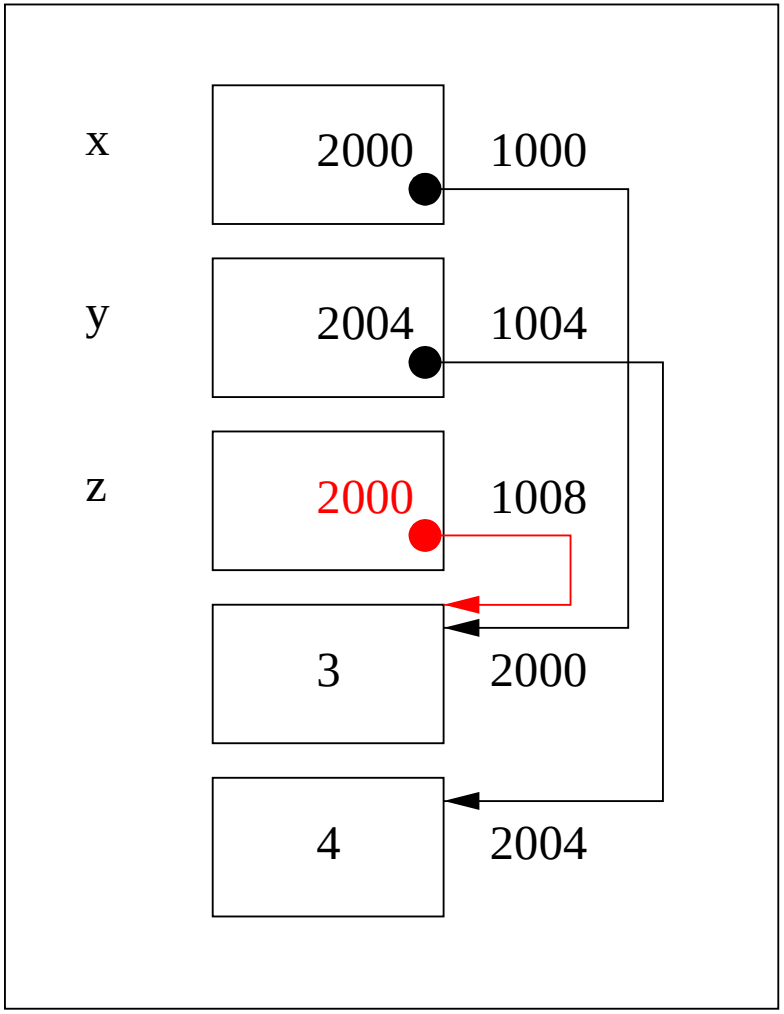
C++ pointer example 2

```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



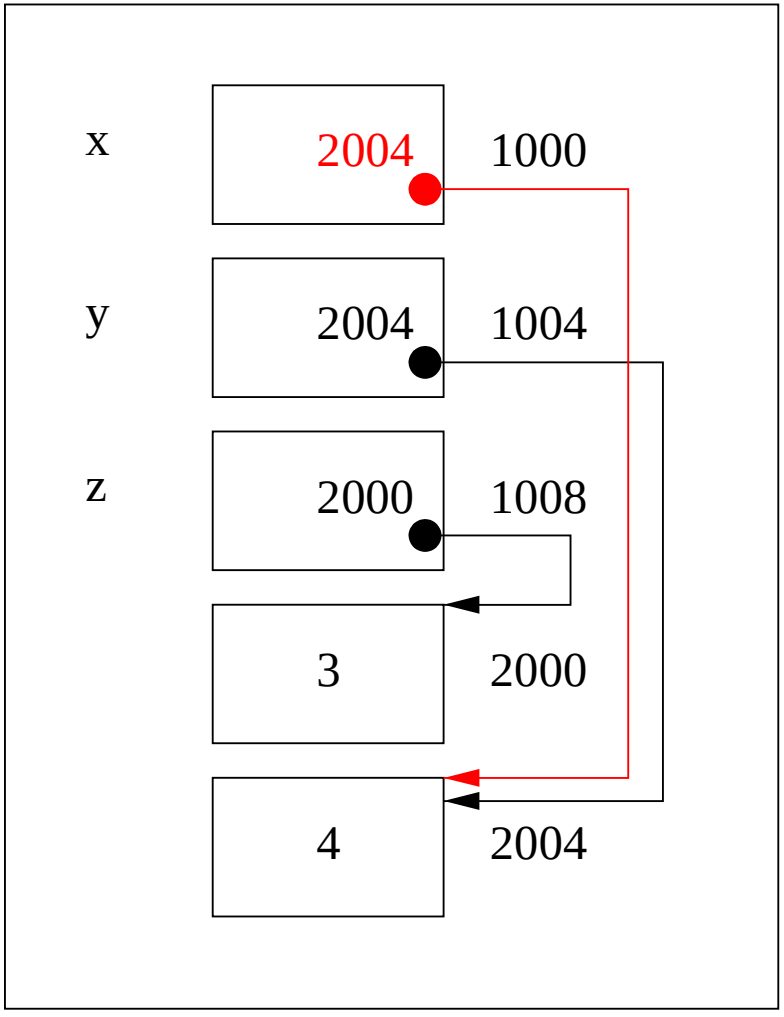
C++ pointer example 2

```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



C++ pointer example 2

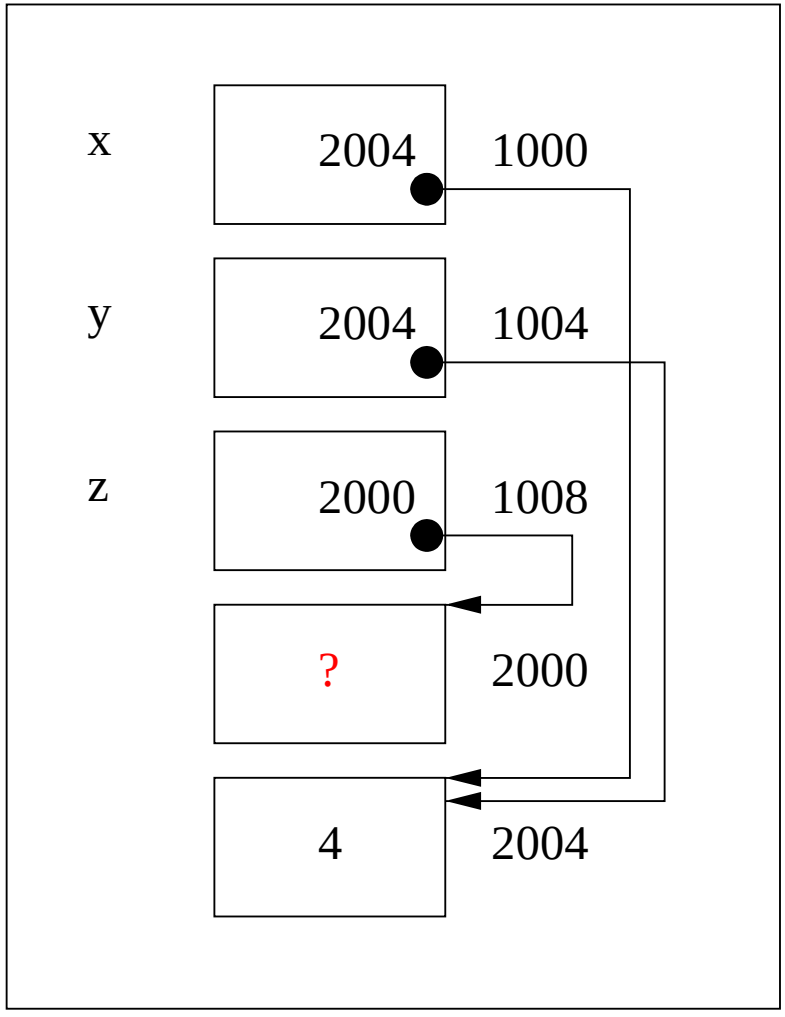
```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



C++ pointer example 2

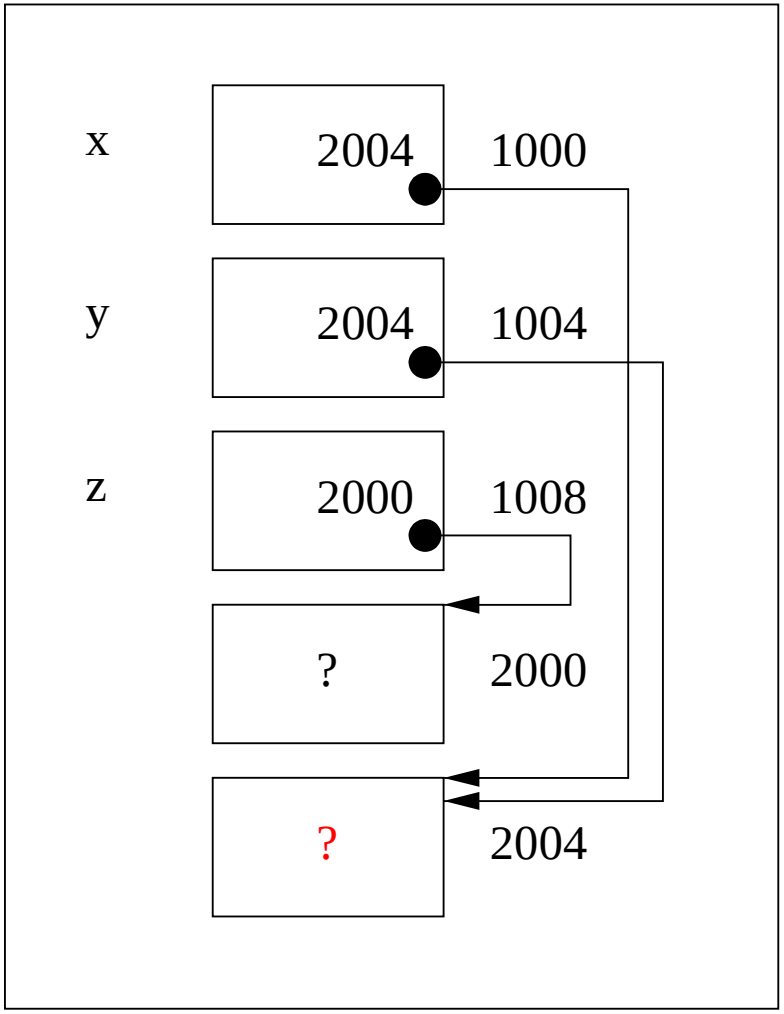
```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```

delete statement deallocates memory that was dynamically allocated



C++ pointer example 2

```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



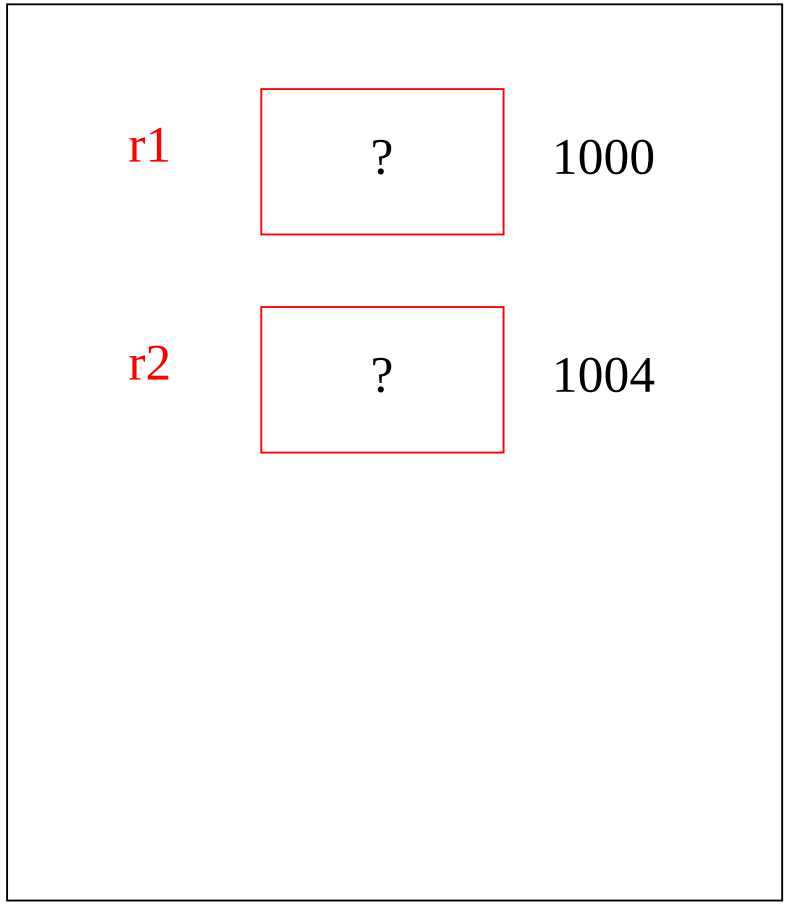
Memory leak

Remember: each `new` statement that is executed must eventually have a corresponding `delete` statement that is executed to deallocate the memory.

If you forget a `delete` statement, your program will have a memory leak. Even though a program with memory leak may not crash, the code is considered incorrect.

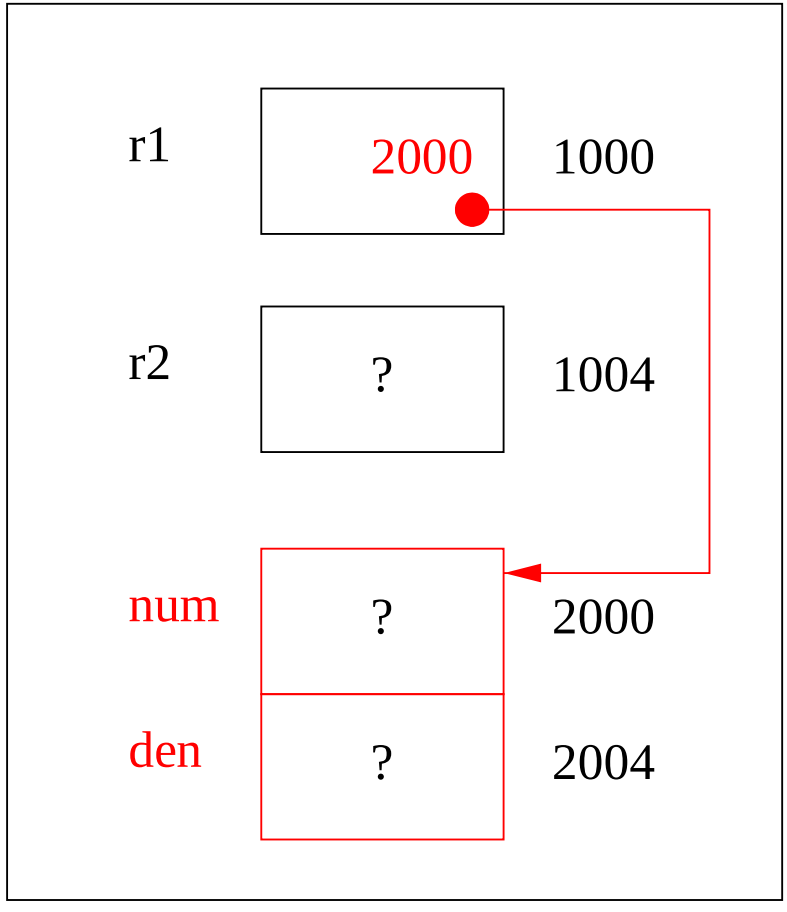
C++ pointer example 3

```
Rational *r1, *r2;
//constructors not called
r1 = new Rational;
r1->set(2, 3);
r2 = r1;
r1->set(1, 3);
delete r1;
```



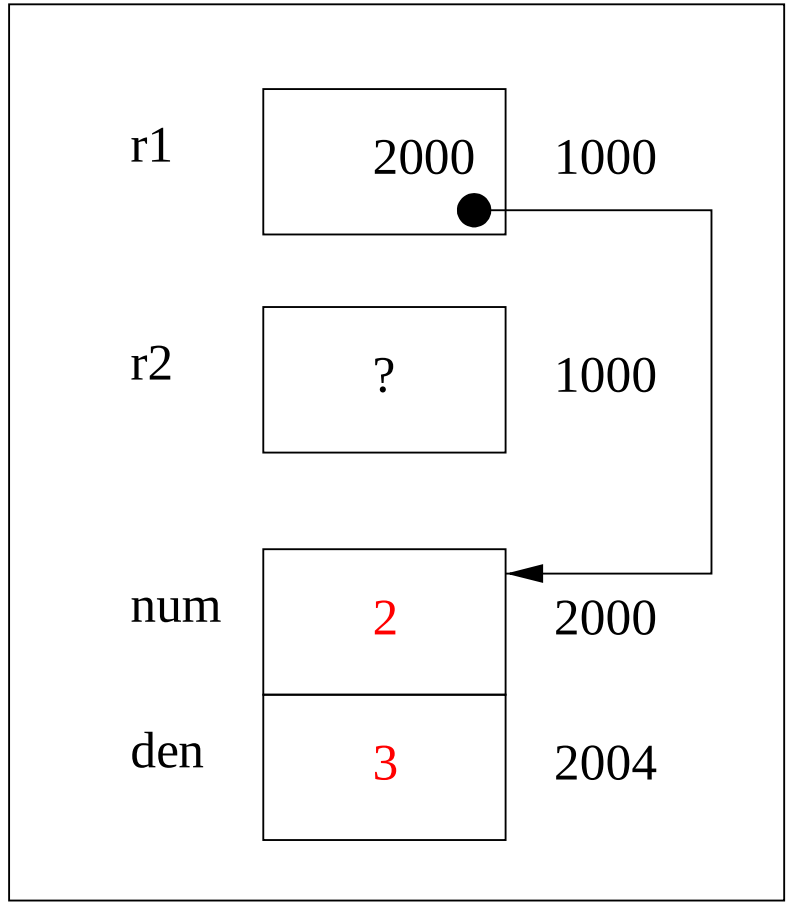
C++ pointer example 3

```
Rational *r1, *r2;
r1 = new Rational;
//constructor is called
r1->set(2, 3);
r2 = r1;
r1->set(1, 3);
delete r1;
```



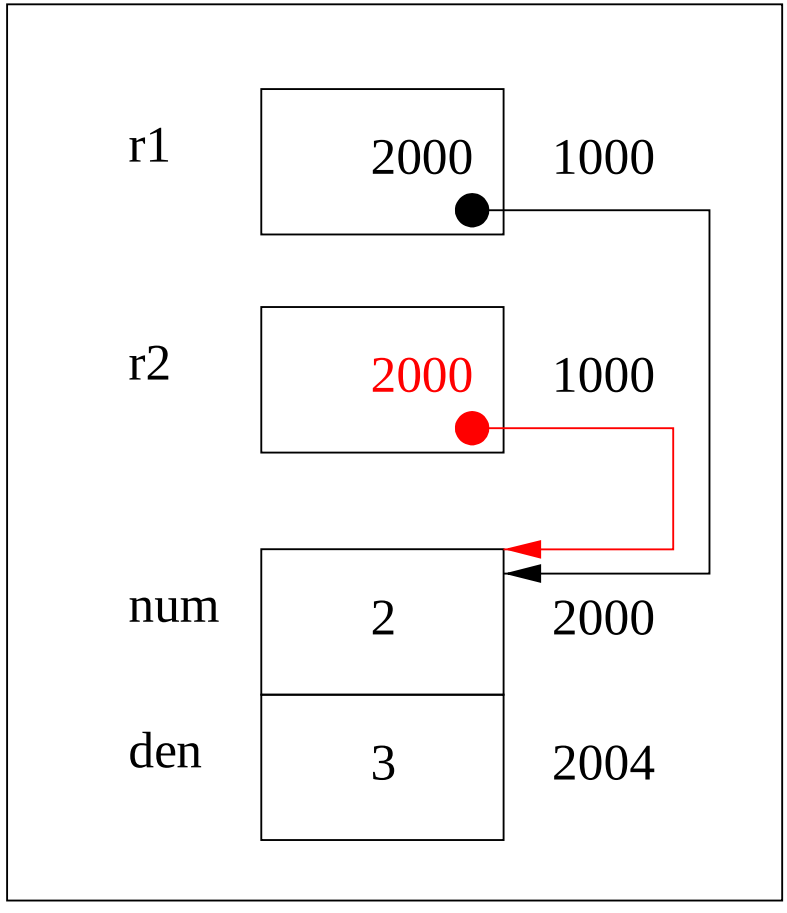
C++ pointer example 3

```
Rational *r1, *r2;
r1 = new Rational;
r1->set(2, 3);
//-> replaced (*r1).set(2,3)
r2 = r1;
r1->set(1, 3);
delete r1;
```



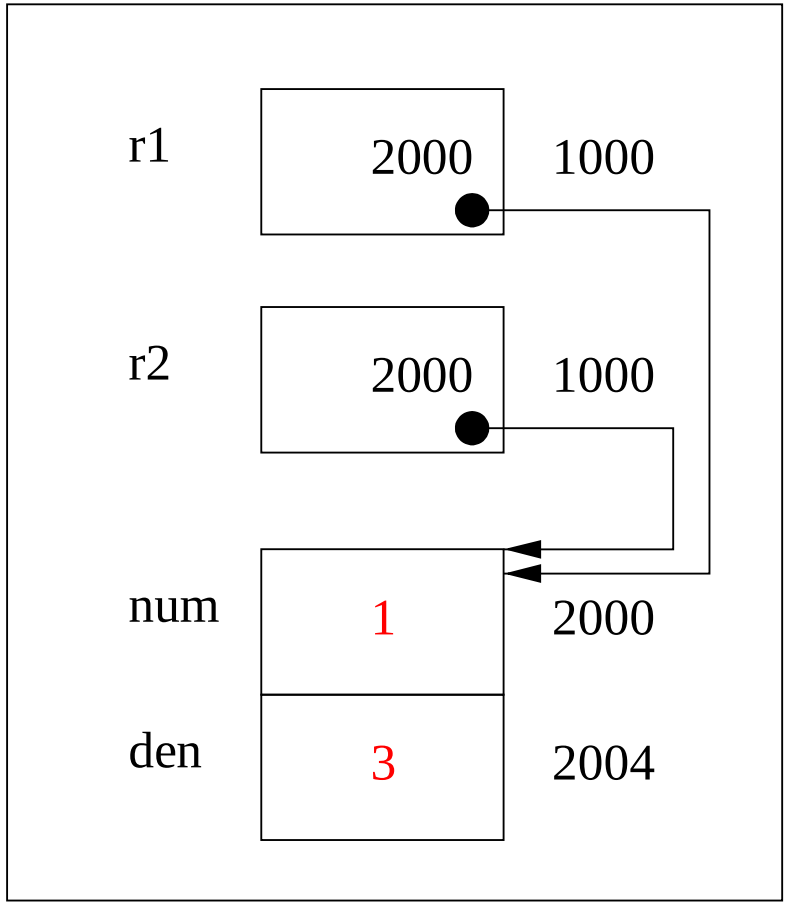
C++ pointer example 3

```
Rational *r1, *r2;
r1 = new Rational;
r1->set(2, 3);
r2 = r1;
r1->set(1, 3);
delete r1;
```



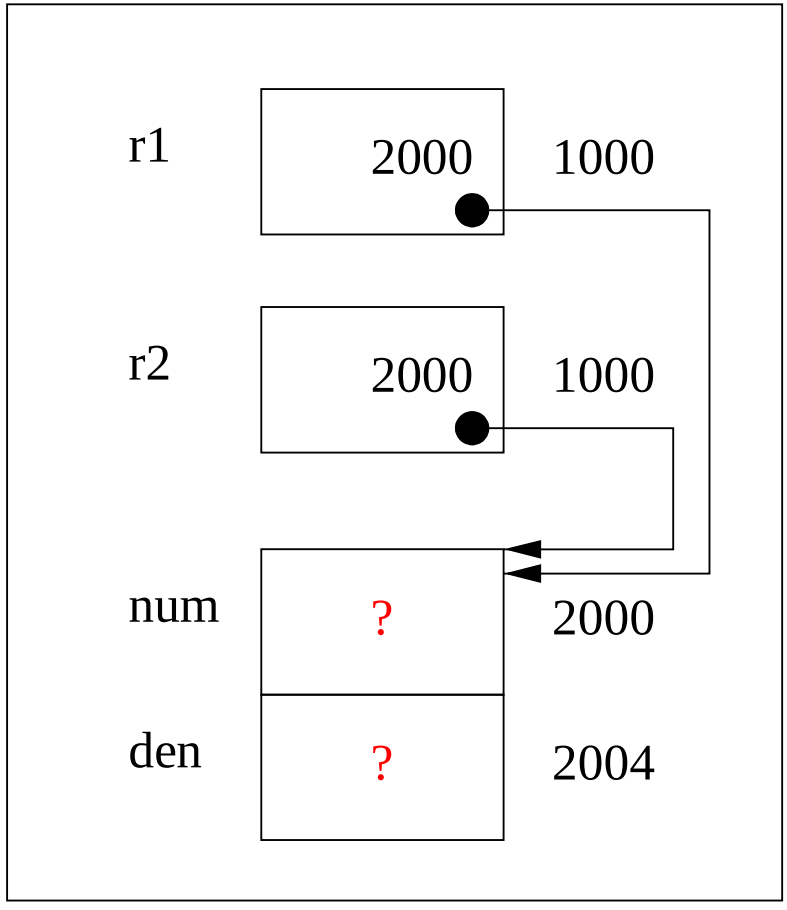
C++ pointer example 3

```
Rational *r1, *r2;
r1 = new Rational;
r1->set(2, 3);
r2 = r1;
r1->set(1, 3);
delete r1;
```



C++ pointer example 3

```
Rational *r1, *r2;
r1 = new Rational;
r1->set(2, 3);
r2 = r1;
r1->set(1, 3);
delete r1;
```



C++ arrays: static arrays

- Consider declaration `int a[20];`
- Internally, `a` is a pointer to the first item in the array, `a[0]`.
- The size is fixed at compile time.
  here, it is 10: `int a[10];`
- A *static array* cannot be expanded to a larger capacity.

C++ arrays: dynamic arrays

- A dynamic array is explicitly declared as a pointer:

```
int *a;
```

- It is given an initial size using the new operator:

```
a = new int[5];
```

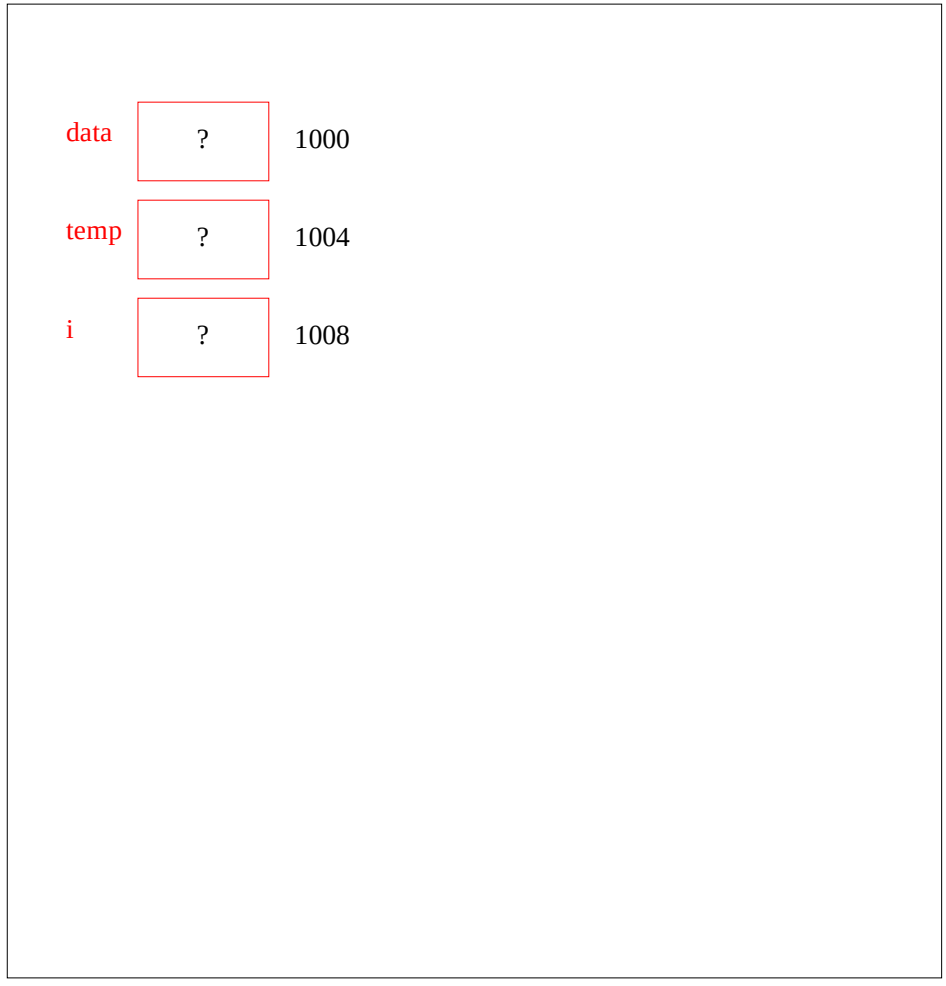
- A dynamic array can be expanded: its items can be copied into a larger area, whose address can be assigned to the original variable.

C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i; }

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i];}

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i; }
delete [] data;
```

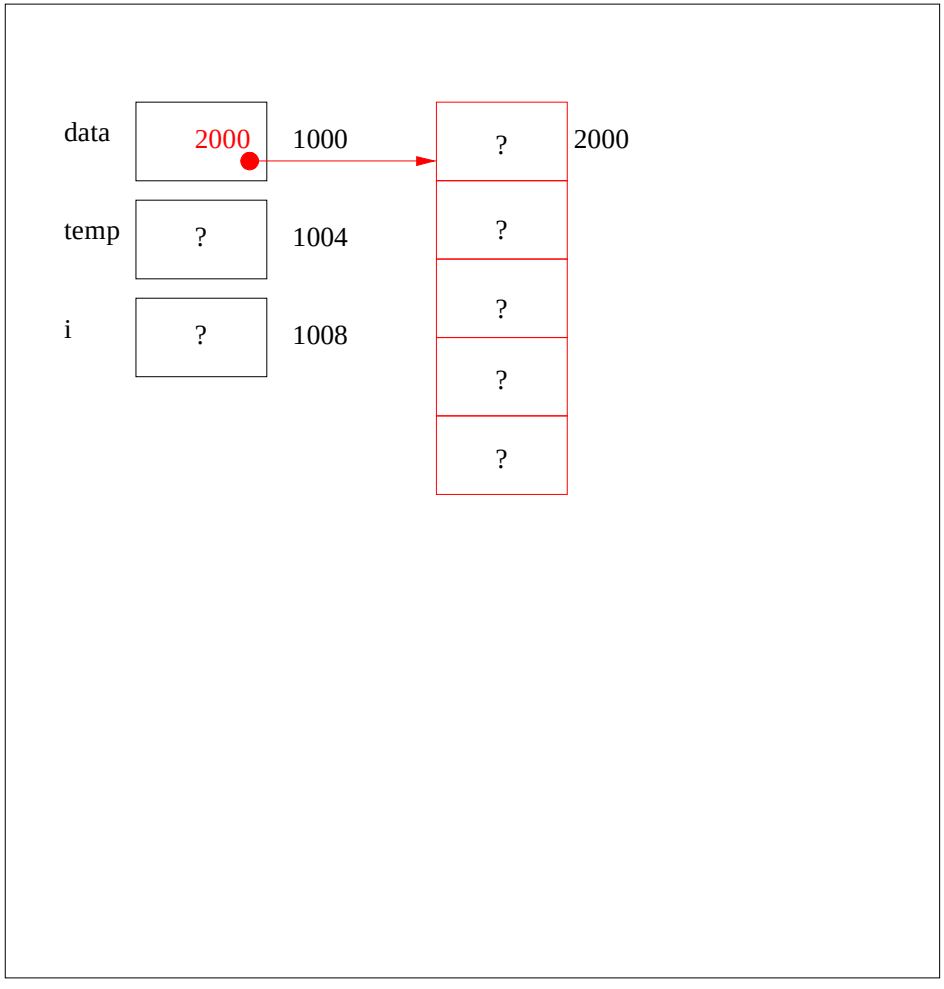


C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i; }

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i];}

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i; }
delete [] data;
```

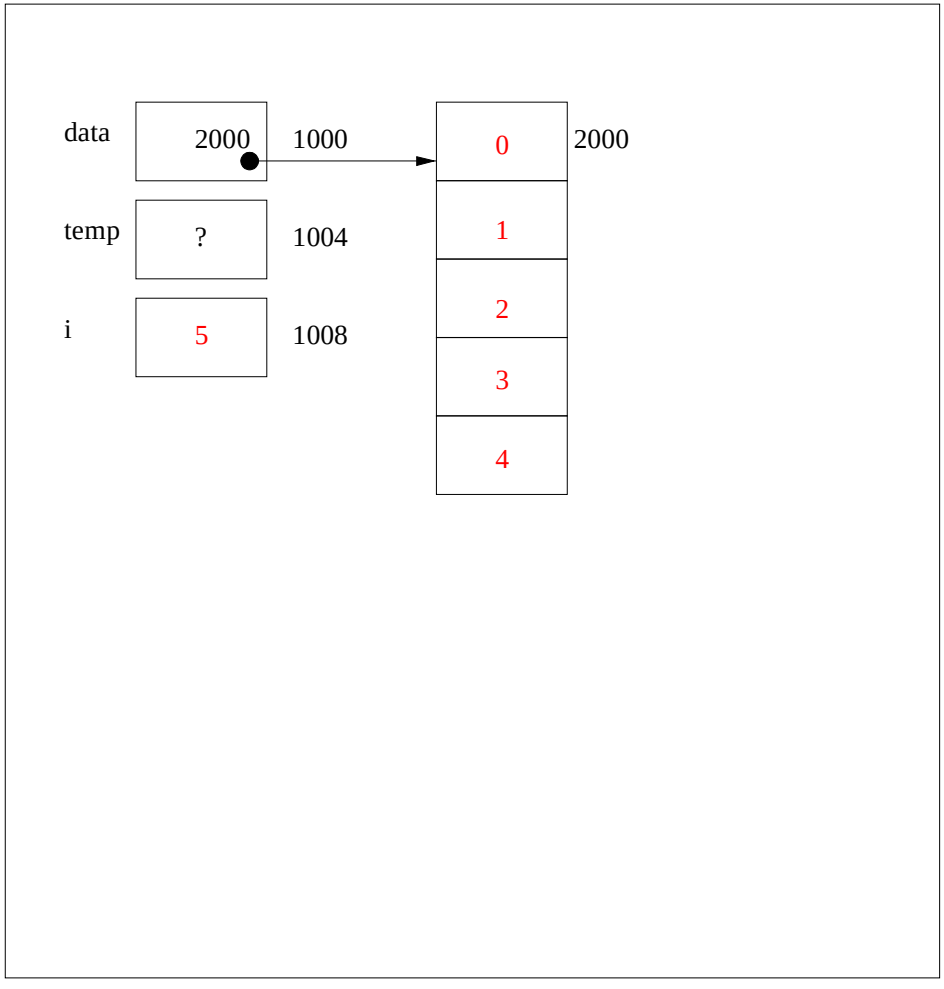


C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i;}

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i]; }

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i;}
delete [] data;
```

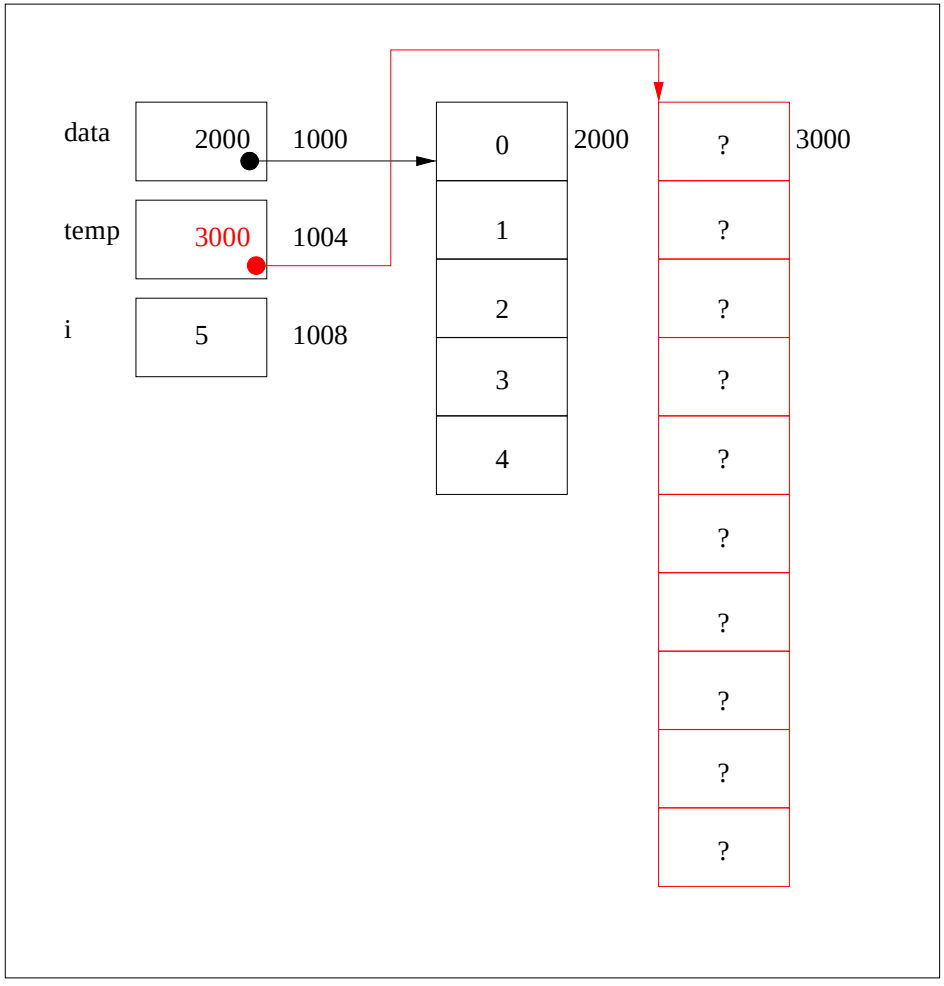


C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i;}

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i]; }

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i;}
delete [] data;
```

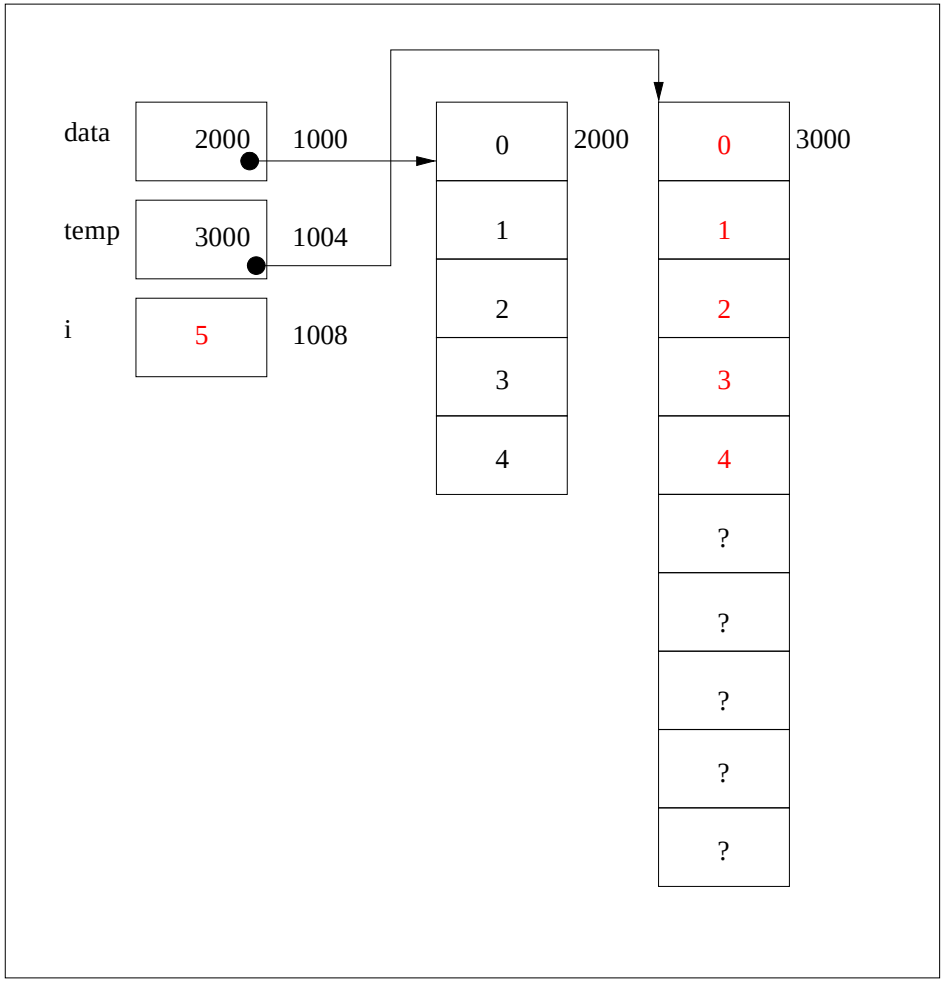


C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i;}

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i];}

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i;}
delete [] data;
```

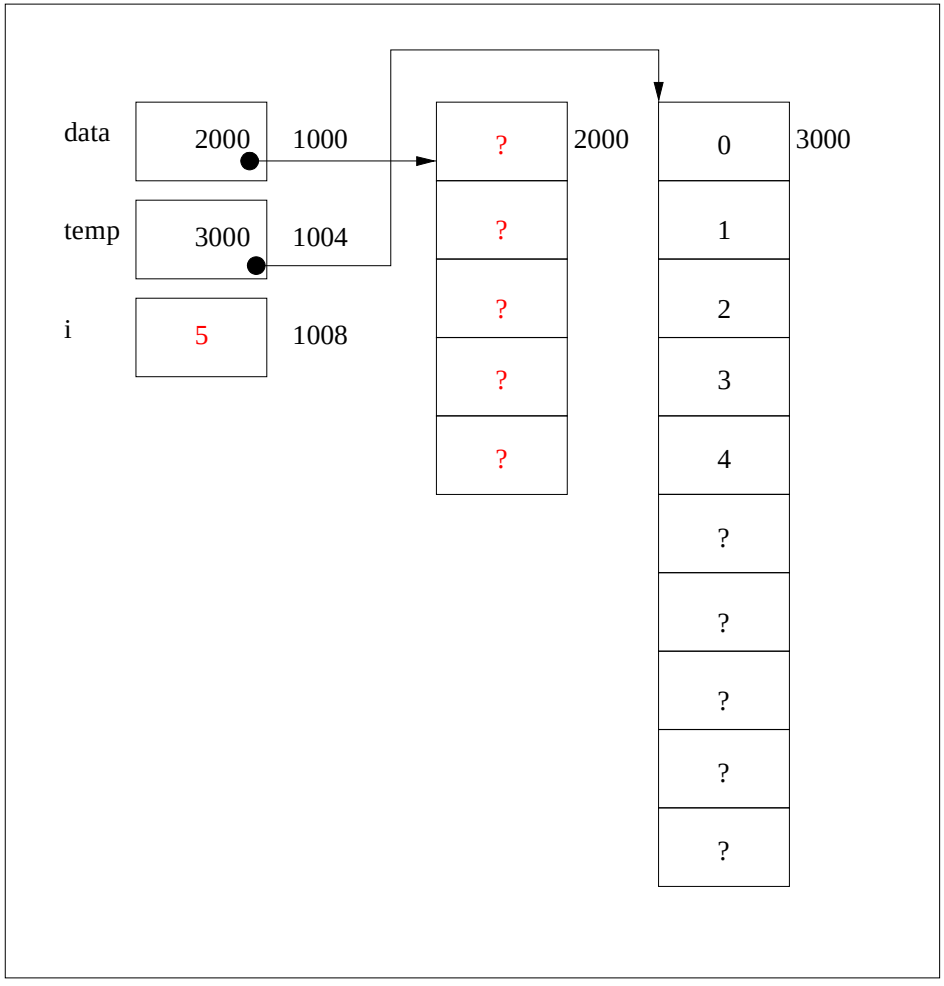


C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i;}

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i];}

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i;}
delete [] data;
```

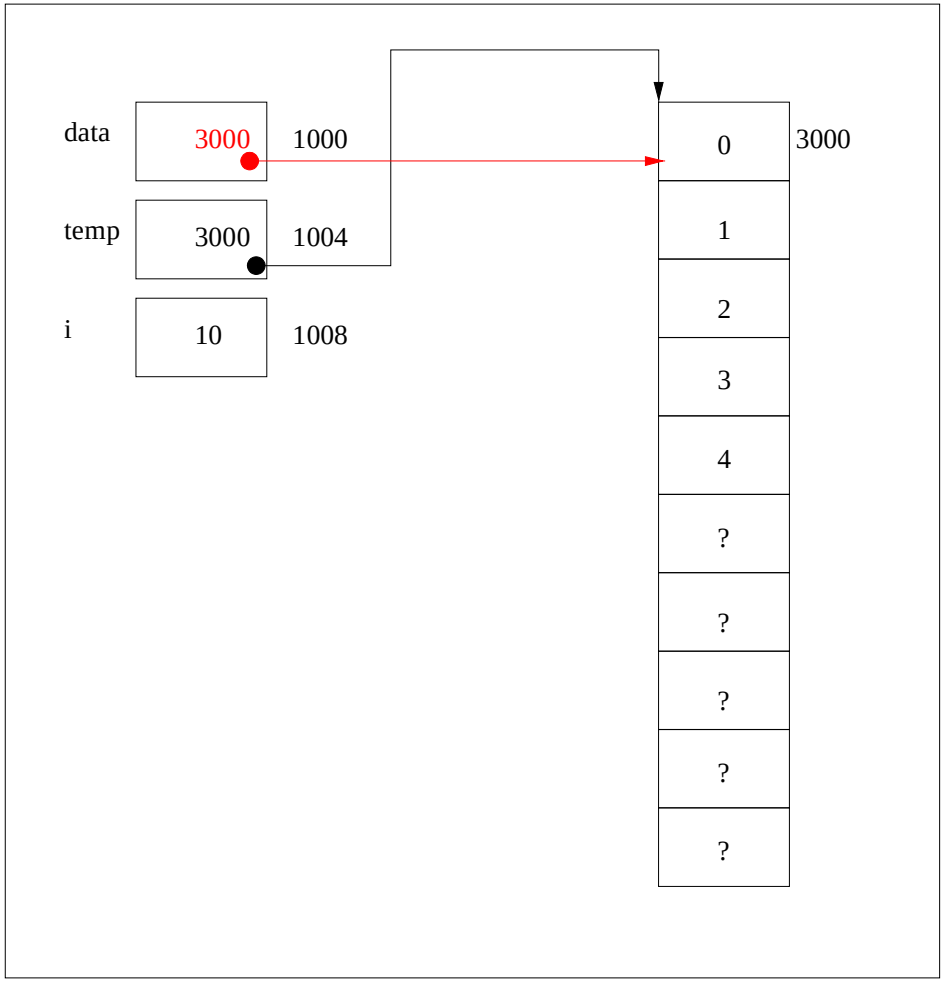


C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i;}

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i];}

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i;}
delete [] data;
```

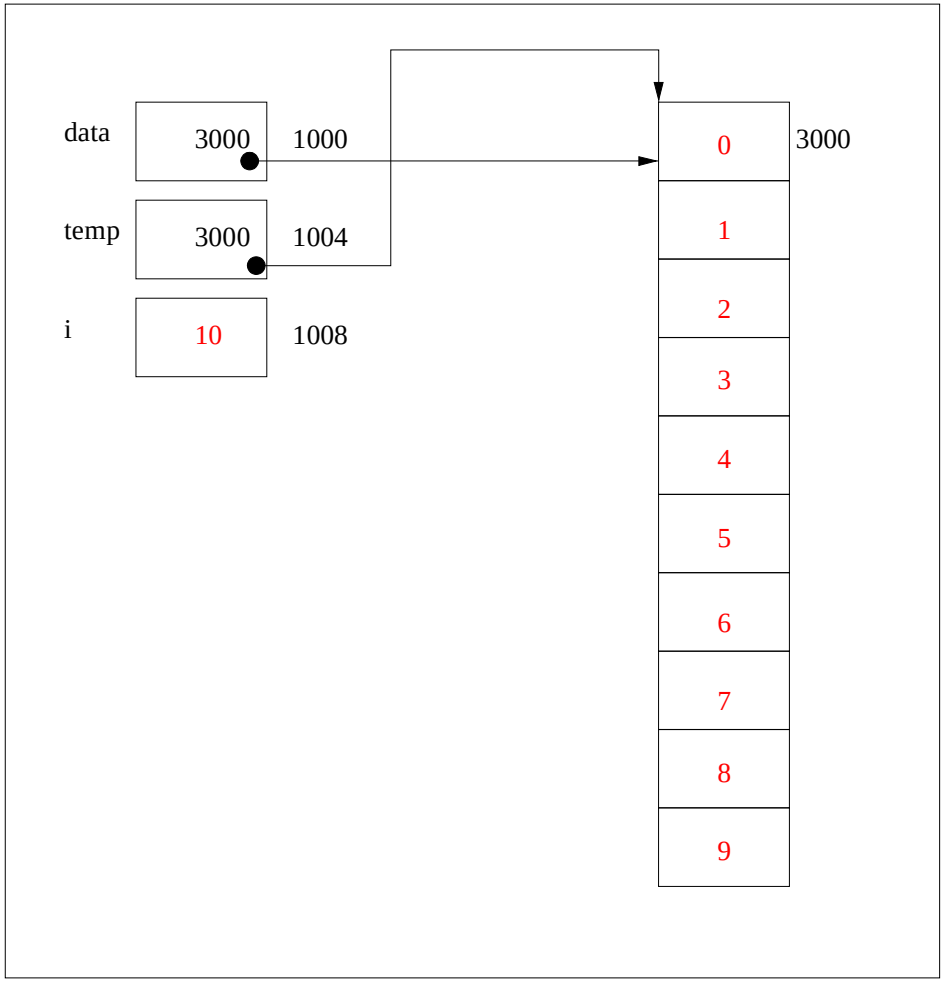


C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i;}

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i];}

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i;}
delete [] data;
```



C++ arrays: dynamic array example

```
int *data, *temp, i;
data = new int[5];
for (i=0; i<5; ++i) {
    data[i] = i;}

temp = new int[10];
for (i=0; i<5; ++i) {
    temp[i] = data[i];}

delete [] data;
data = temp;
for (i=0; i<10; ++i) {
    data[i] = i;}
delete [] data;
```