CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

Outline

1. Chapter 10: C++ Dynamic Memory
   - Introduction
   - C++ Pointers
   - Dynamic Arrays
   - In-class work

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.

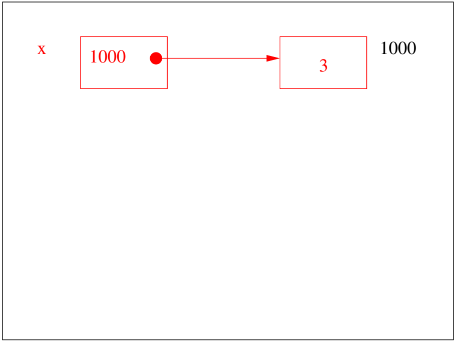
Storing Variables In Python and C++

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.

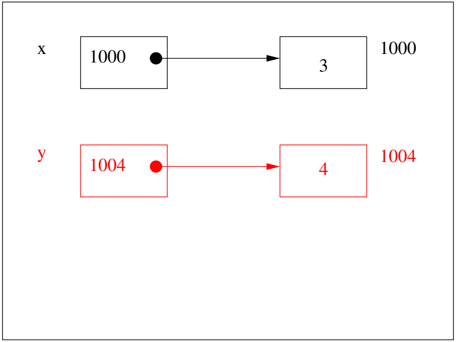
Python Memory Example

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



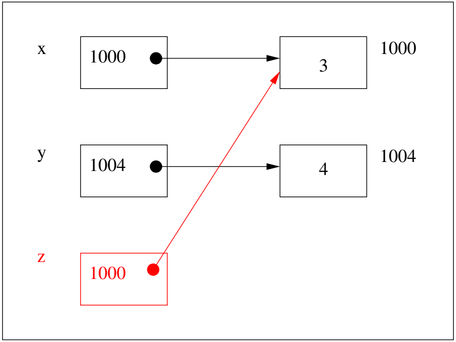
Python Memory Example

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



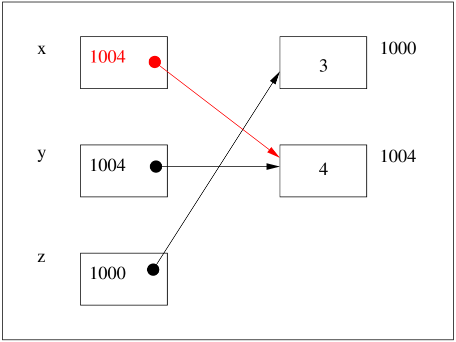
Python Memory Example

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



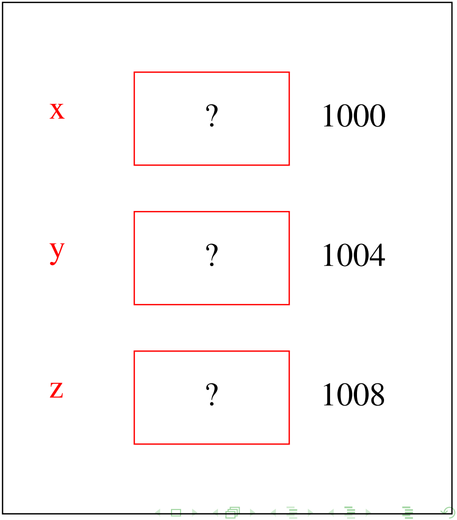
Python Memory Example

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



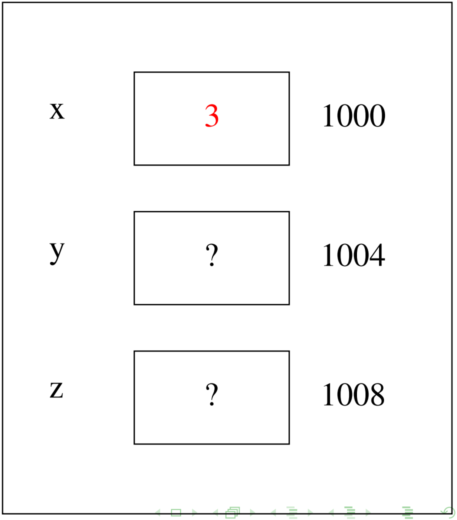
C++ Memory Example

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



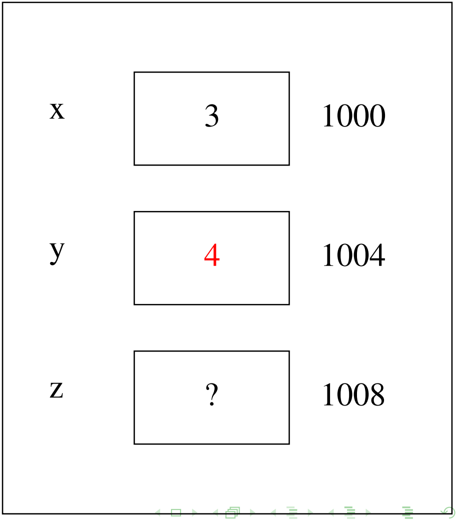
C++ Memory Example

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



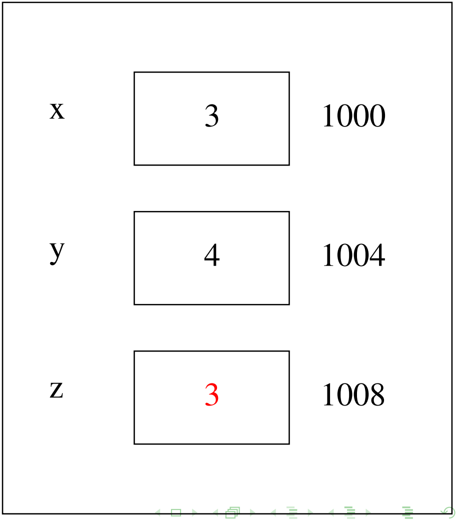
C++ Memory Example

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



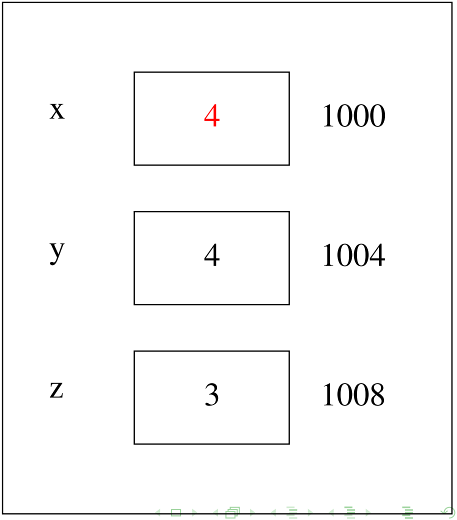
C++ Memory Example

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



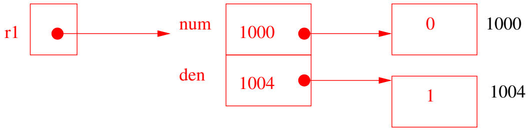
C++ Memory Example

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



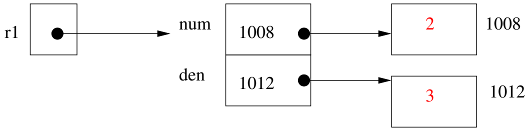
Python Object Example

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



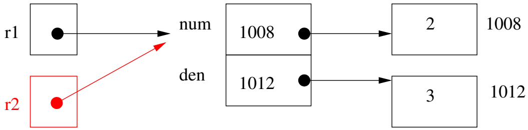
Python Object Example

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



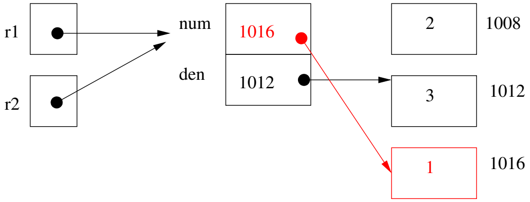
Python Object Example

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



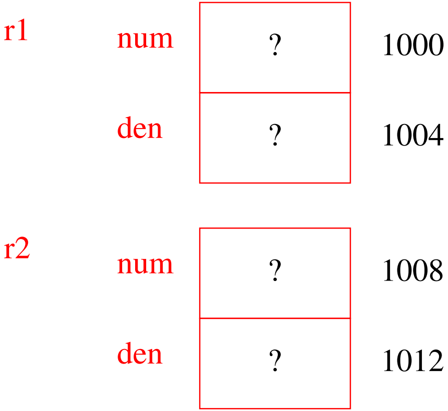
Python Object Example

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



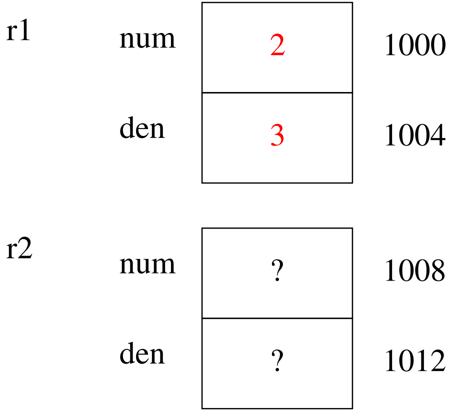
C++ Object Example

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



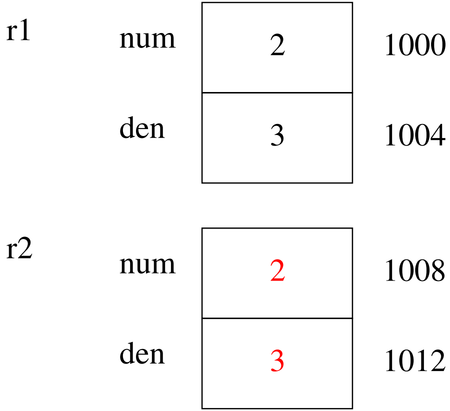
C++ Object Example

```
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);
```

r1	num	1	1000
	den	3	1004
r2	num	2	1008
	den	3	1012

Pointer Syntax

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.

Pointer Syntax

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.

Pointer Syntax

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.

Pointer Syntax

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.

Pointer Syntax

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.

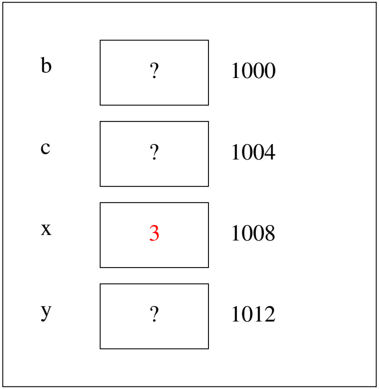
C++ Pointer Example 1

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

b	?	1000
c	?	1004
x	?	1008
y	?	1012

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;
```



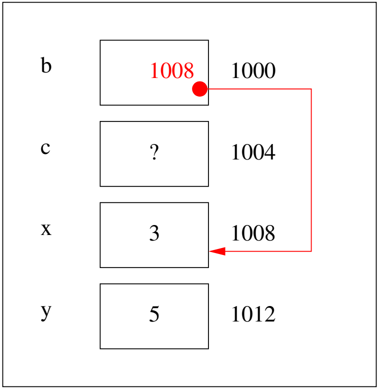
C++ Pointer Example 1

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

b	?	1000
c	?	1004
x	3	1008
y	5	1012

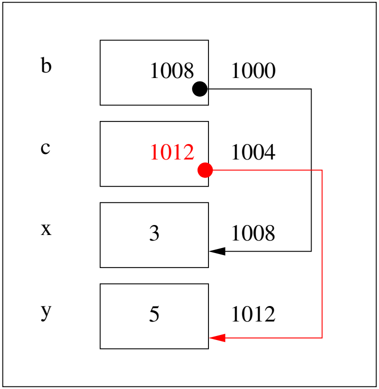
C++ Pointer Example 1

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



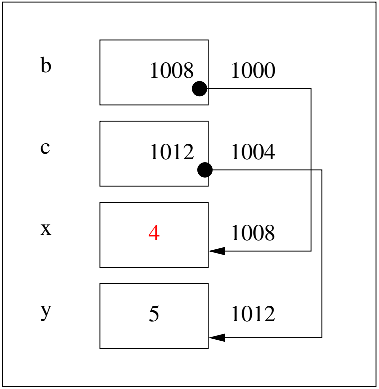
C++ Pointer Example 1

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



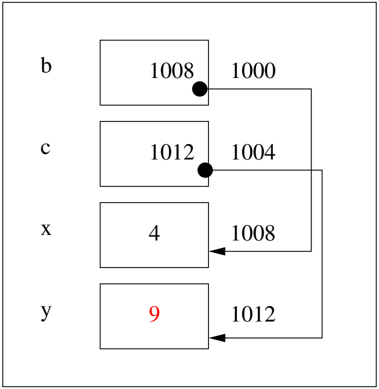
C++ Pointer Example 1

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



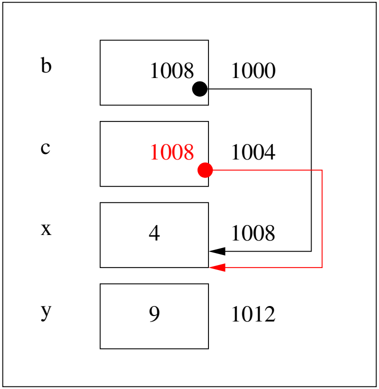
C++ Pointer Example 1

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



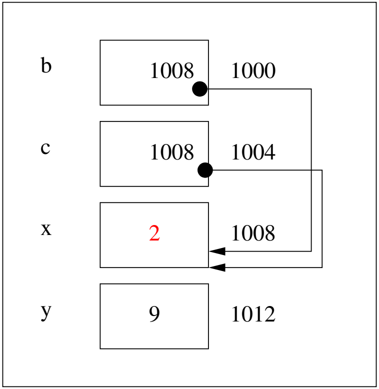
C++ Pointer Example 1

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



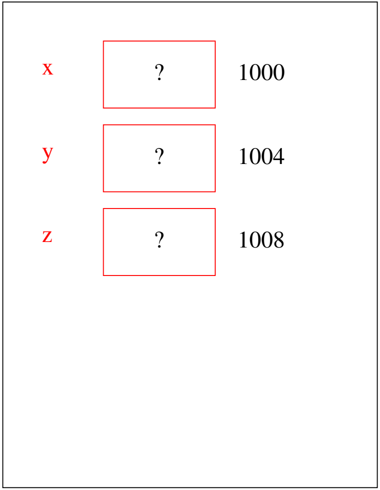
C++ Pointer Example 1

```
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

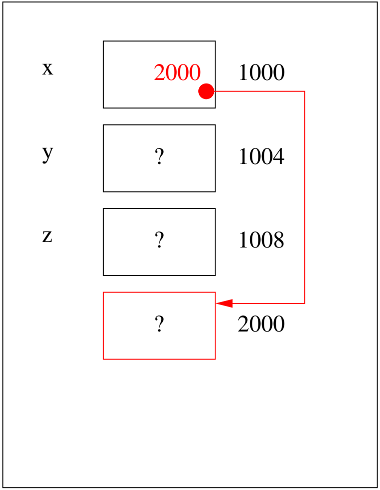
```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



C++ Pointer Example 2

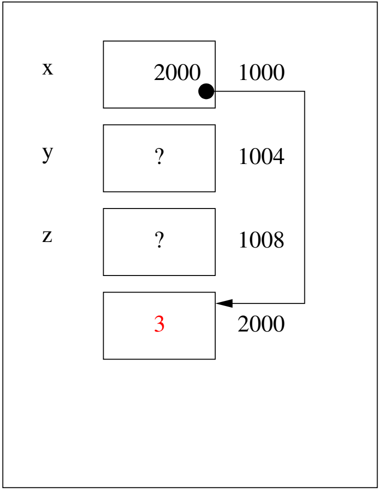
```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```

new statement allocates dynamic memory and returns the starting address.



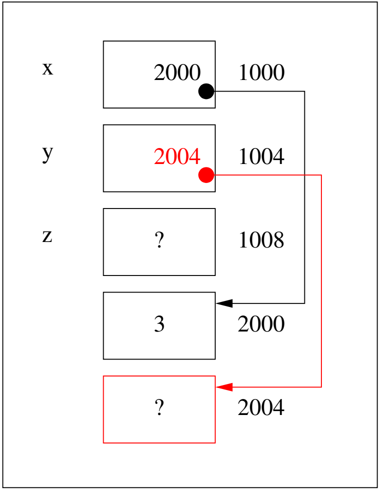
C++ Pointer Example 2

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



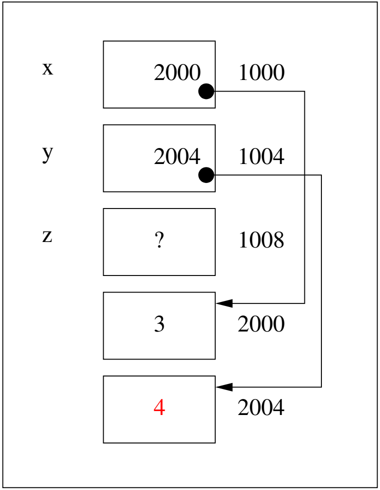
C++ Pointer Example 2

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



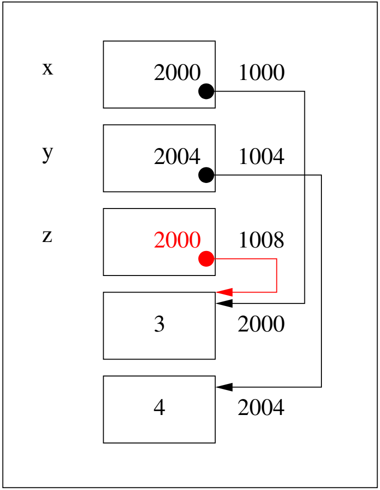
C++ Pointer Example 2

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



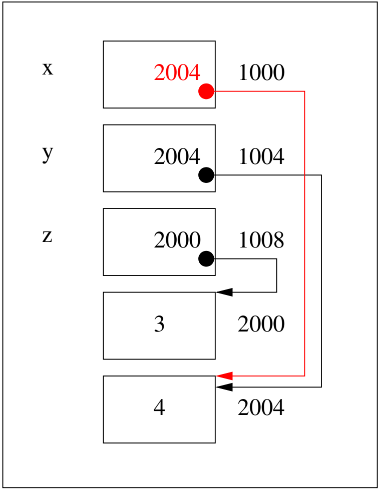
C++ Pointer Example 2

```
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

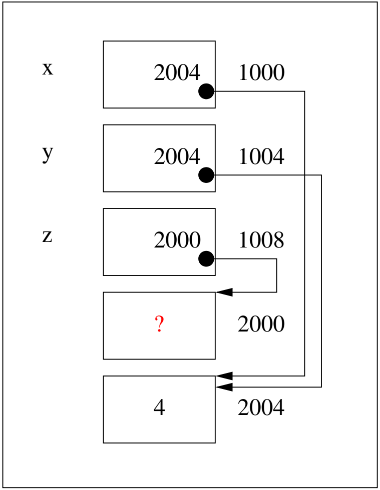
```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```



C++ Pointer Example 2

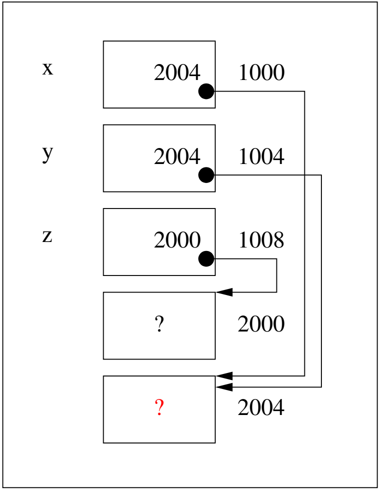
```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;
delete z;
delete y;
```

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

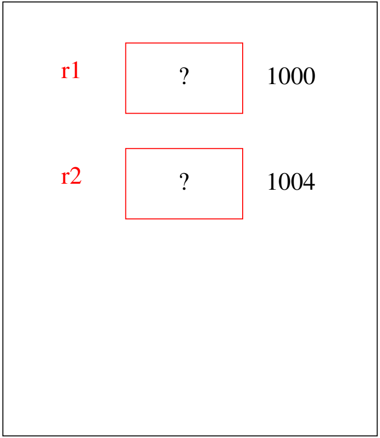
Remember: each `new` statement that is executed must eventually have a corresponding `delete` statement that is executed to deallocate the memory.

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.

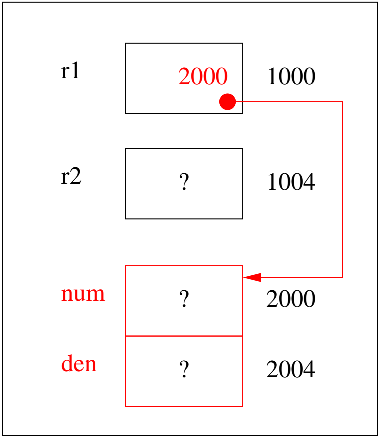
C++ Pointer Example 3

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



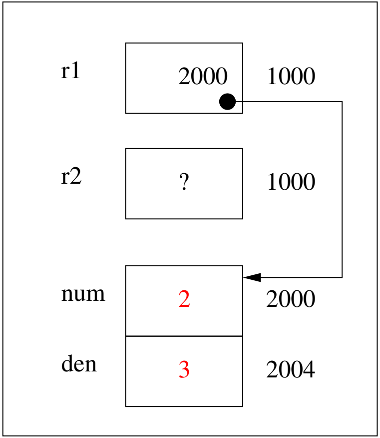
C++ Pointer Example 3

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



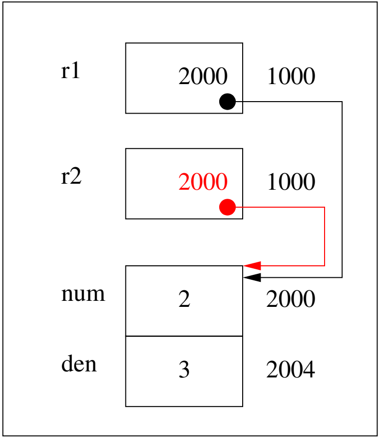
C++ Pointer Example 3

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



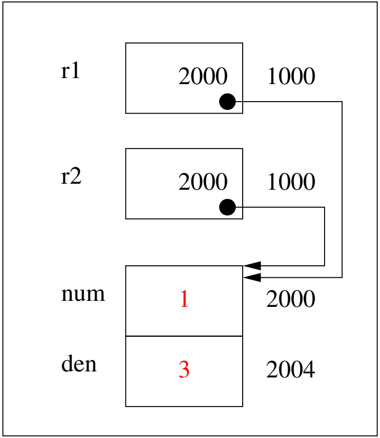
C++ Pointer Example 3

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



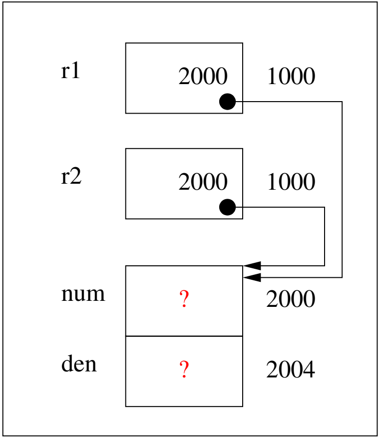
C++ Pointer Example 3

```
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

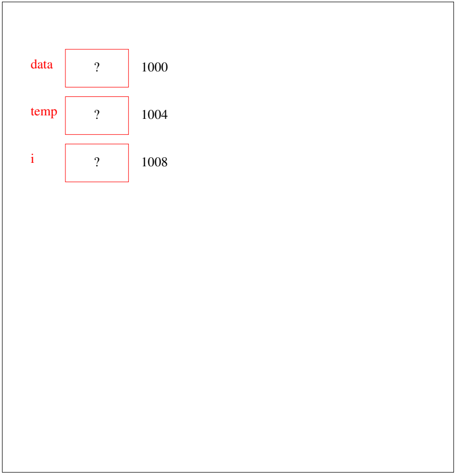
- A dynamic array is explicitly declared as a pointer:
  `int *a;`
- It is given an initial size using the **new** operator:
  `a = new int[5];`
- A dynamic array can be expanded: Its items can be copied into a larger area, whose address can be assigned to the original variable.

C++ 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;
```

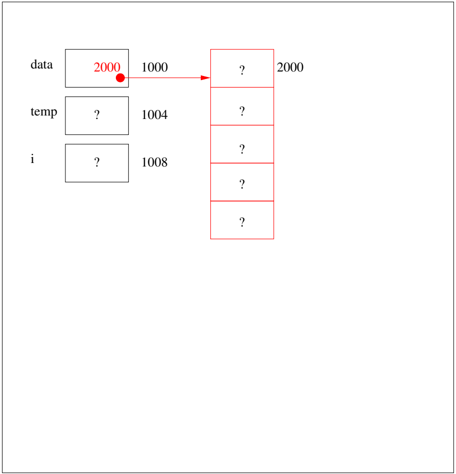


C++ Dynamic Array Example

```
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;
```

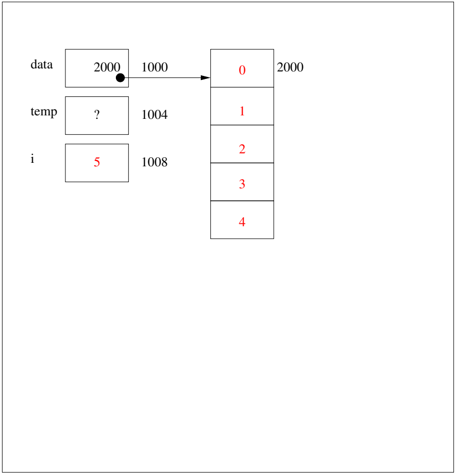


C++ Dynamic Array Example

```
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;
```

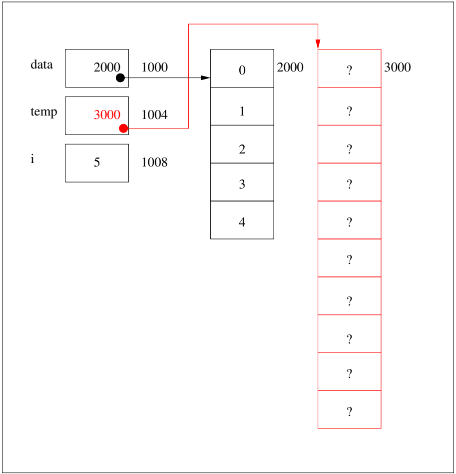


C++ Dynamic Array Example

```
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;
```

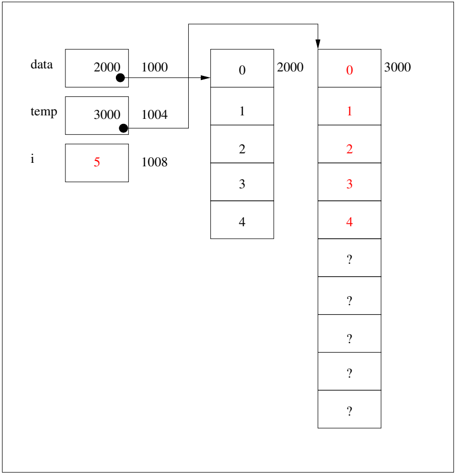


C++ Dynamic Array Example

```
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;
```

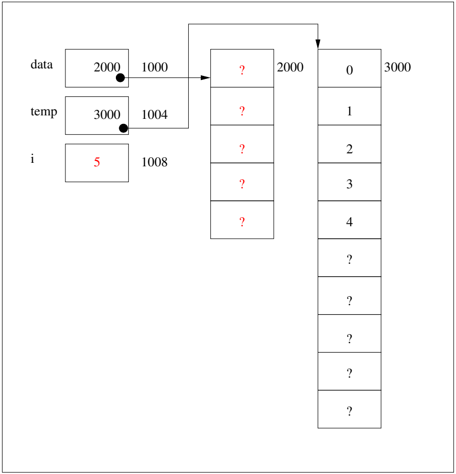


C++ Dynamic Array Example

```
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;
```

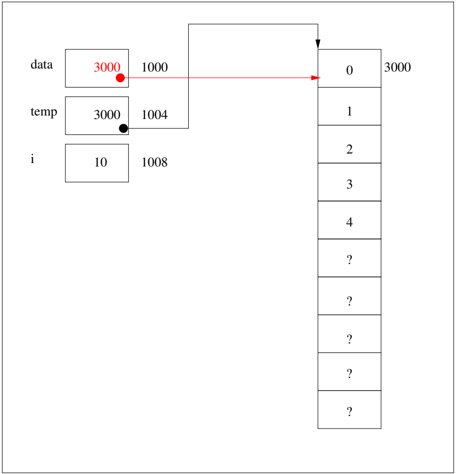


C++ Dynamic Array Example

```
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;
```

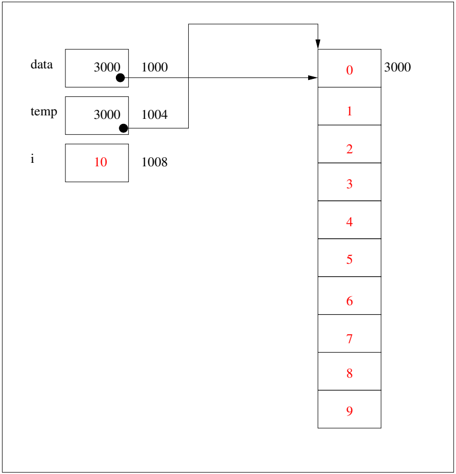


C++ Dynamic Array Example

```
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;
```

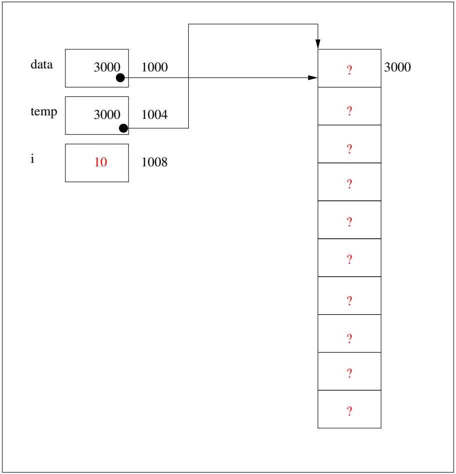


C++ Dynamic Array Example

```
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;
```



In-class work

1) Look at the following code and draw memory representation at stages 1, 2, and 3.

```
char *a, *b, t = 'f',
z='o'; // stage 1
a = new char;
*a = t;
b = &z; // stage 2
*b = 'g';
b = &z;
*b = 'h'; // stage 3
delete a;
delete b;
```

Do we have a memory leak?

In-class work

2) Write a C++ program that will do the following:

- Create two integer pointers **a** and **b** (they will be pointing to two arrays of integer values), then
- dynamically allocate space for an array of 100 integer values, the pointer **a** will be referencing it, then
- fill this array with odd numbers starting with 3, then
- dynamically allocate space for another array, with 200 integer values the pointer **b** will be referencing it, then
- fill this array with even numbers starting with 4, then
- add the values from the first array to the corresponding values of the second array (100 additions should be performed, the new values should be stored in the second array), then
- *deallocate* the first array pointer **a** is referencing, then
- display the values of the second array and *deallocate* it as well.