Memory Model

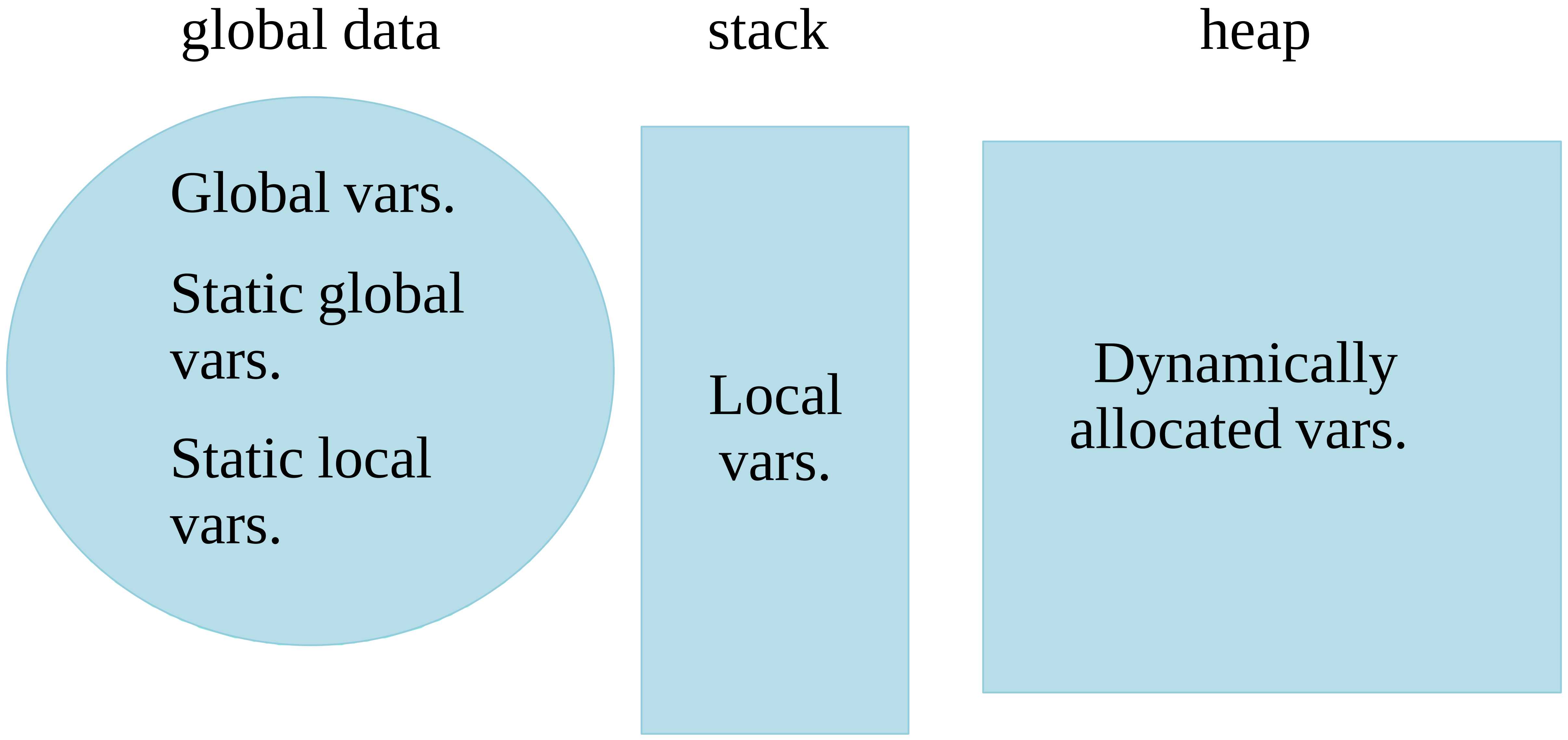
Object-Oriented Programming with C++

Zhaopeng Cui

What are these variables?

```
int i; // global vars.
string str;
static int j; // static global vars.

void f()
{
 int k; // local vars.
 static int l; // static local vars.

 int *p = malloc(sizeof(int)); // allocated vars.
}
```

Where are they in memory?



Global vars

- vars defined outside any functions
- can be shared between .cpp files
- extern

extern

- **extern** is a declaration says there will be such a variable somewhere in the whole program
- “ such a” means the type and the name of the variable
- global variable is a definition, the place for that variable

static

- static global variable inhibits access from outside the .cpp file
- so as the static function

static local var

- static local variable keeps value in between visits to the same function
- is to be initialized at its first access

static

- for global stuff:
  - access restriction

- for local stuff:
  - persistence

Pointers to Objects

Pointers to Objects

```
string s = "hello";

string* ps = &s;
```

Operators with Pointers

- &: get address

```
ps = &s;
```

- *: get the object

```
(*ps).length()
```

- ->: call the function

```
ps->length()
```

Two Ways to Access

- string s;
    - s is the object itself
    - At this line, object s is created and initialized

- string *ps;
    - At this line, ps is a pointer to an object
    - At this line, the object ps points to is not known yet.

Assignment

```
string s1, s2;
s1 = s2;

string *ps1, *ps2;
ps1 = ps2;
```

reference

Declaring references

- References are a new data type in C++

```
char c;         // a character
char* p = &c;   // a pointer to a character
char& r = c;    // a reference to a character
```

- Local or global variables
  - type& refname = name;
  - For ordinary variables, the initial value is required
- In parameter lists and member variables
  - type& refname
  - Binding defined by caller or constructor

Declaring references

- Declares a new name for an existing object

```
int X = 47;
int &Y = X;
// Y is a reference to X, X and Y now refer to
// the same variable

cout << "Y = " << Y; // prints Y = 47
Y = 18;
cout << "X = " << X; // prints X = 18
```

Rules of references

- References must be initialized when defined
- Initialization establishes a binding
    - In definition

```
int x = 3 ;
int& y = x;
const int& z = x;
```

- As a function argument

```
void f (int& x) ;
f(y); // initialized when function is called
```

Declaring references

- Bindings don’t change at run time, unlike pointers
- Assignment changes the object referred-to

```
int& y = x;
y = z; // Change value of x to value of z.
```

- The target of a reference must have a location!

```
void func (int &) ;
func (i * 3); // Warning or Error!
```

Restrictions

- No references to references
- No pointers to references, but reference to pointer is ok.

```
int&* p; // illegal
```

```
void f(int*& p); // ok
```

- No arrays of references

Pointers vs. References

- References
  - can't be null
  - can't change to a new "address" location
  - are dependent on an existing variable, they are an alias for an variable

- Pointers
  - can be set to null
  - can change to point to a different address
  - pointer is independent of existing objects

dynamically allocated memory

Dynamic memory allocation

- new

```
new int;
new Stash;
new int[10];
```

- delete

```
delete p;
delete[] p;
```

new and delete

- Similar to malloc, **new** is the way to allocate memory as a program runs. Pointers become the only access to that memory.

- Similar to free, **delete** enables you to return memory to the memory pool when you are finished with it

- Besides that, **new** and **delete** ensure the right calling of Ctor/Dtor for objects.

Dynamic Arrays

- The **new** operator returns the address of the first element of the block

```
int *psome = new int[10];
```

- The presence of the brackets tells the program that it should free the whole array, not just the element

```
delete[] psome;
```

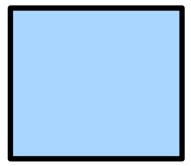
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];
```

```
Student *q=new Student();

Student *r=new Student[10];
```

The new-delete Mechanism

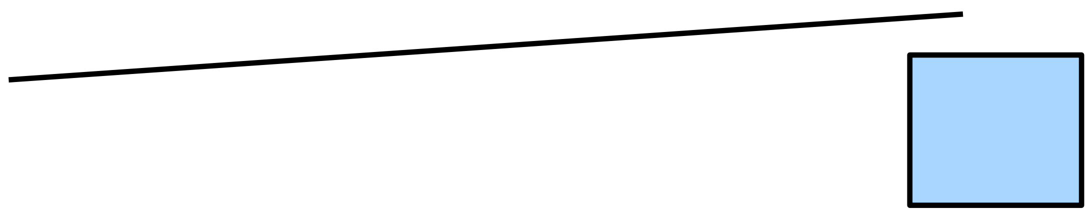
```
int *p=new int;

int *a=new int[10];
```



```
Student *q=new Student();

Student *r=new Student[10];
```

The new-delete Mechanism

```
int *p=new int;
```

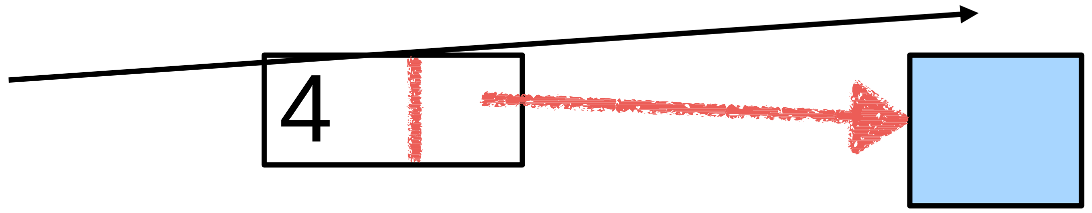


```
int *a=new int[10];
```

```
Student *q=new Student();

Student *r=new Student[10];
```

The new-delete Mechanism

```
int *p=new int;
```

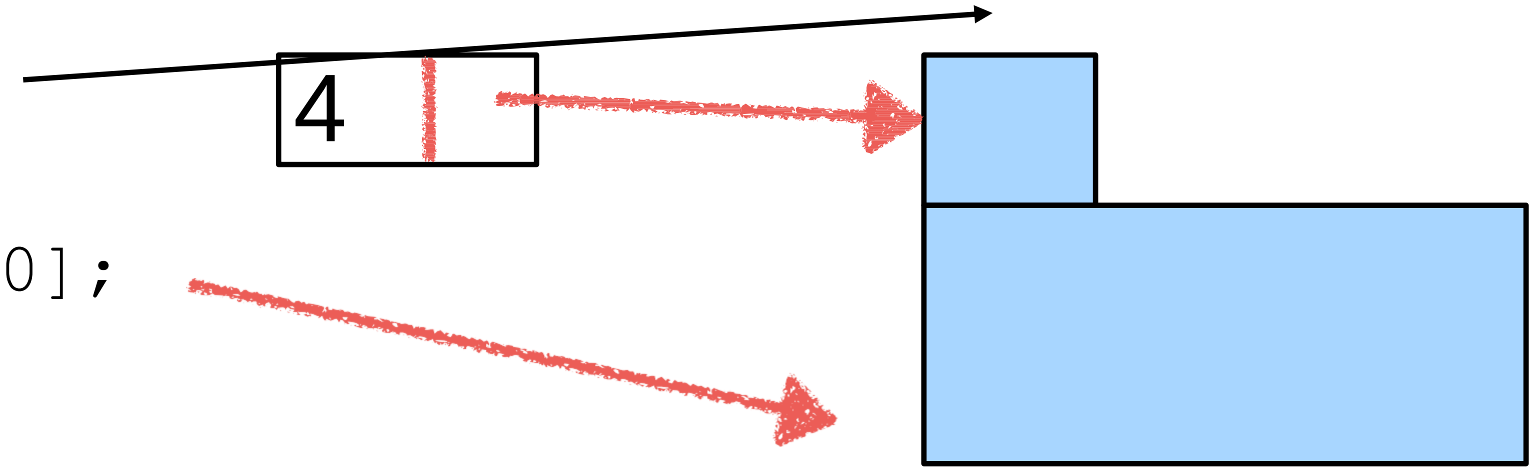


```
int *a=new int[10];
```

```
Student *q=new Student();

Student *r=new Student[10];
```

The new-delete Mechanism

```
int *p=new int;
```

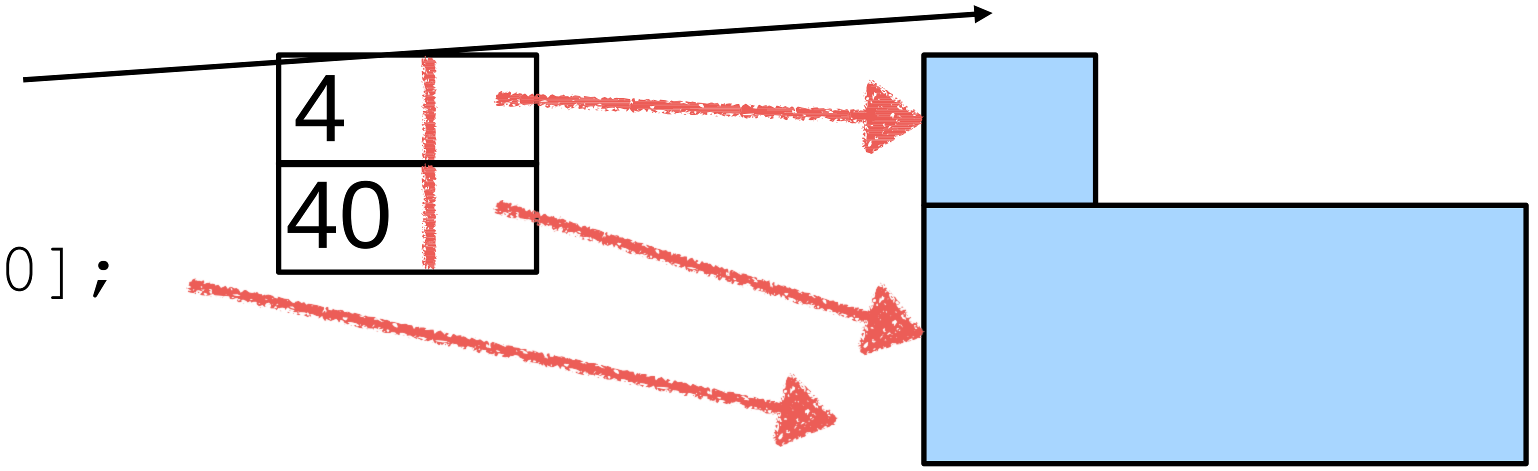


```
int *a=new int[10];
```

```
Student *q=new Student();

Student *r=new Student[10];
```

The new-delete Mechanism

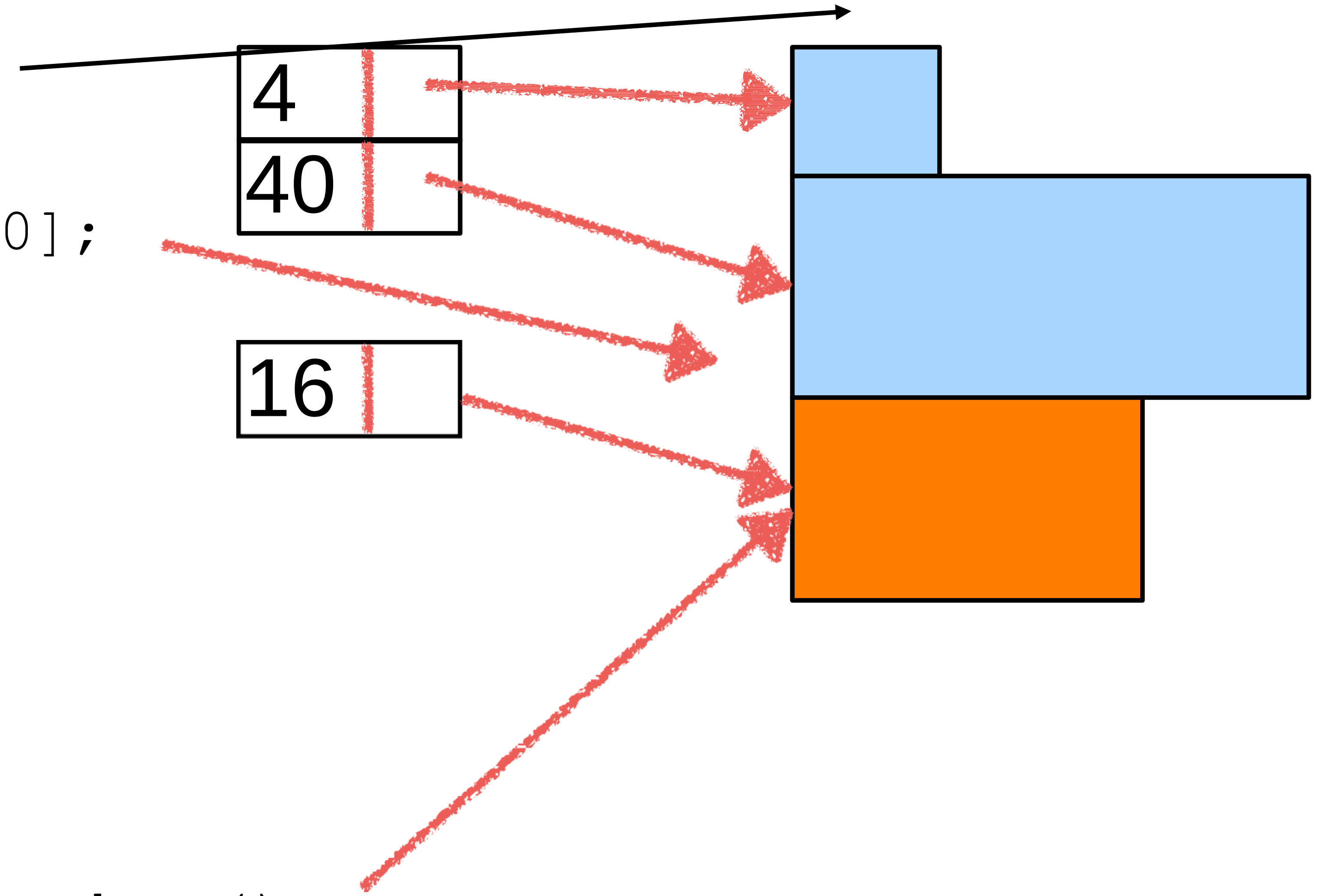
```
int *p=new int;

int *a=new int[10];
```



```
Student *q=new Student();

Student *r=new Student[10];
```

The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];
```

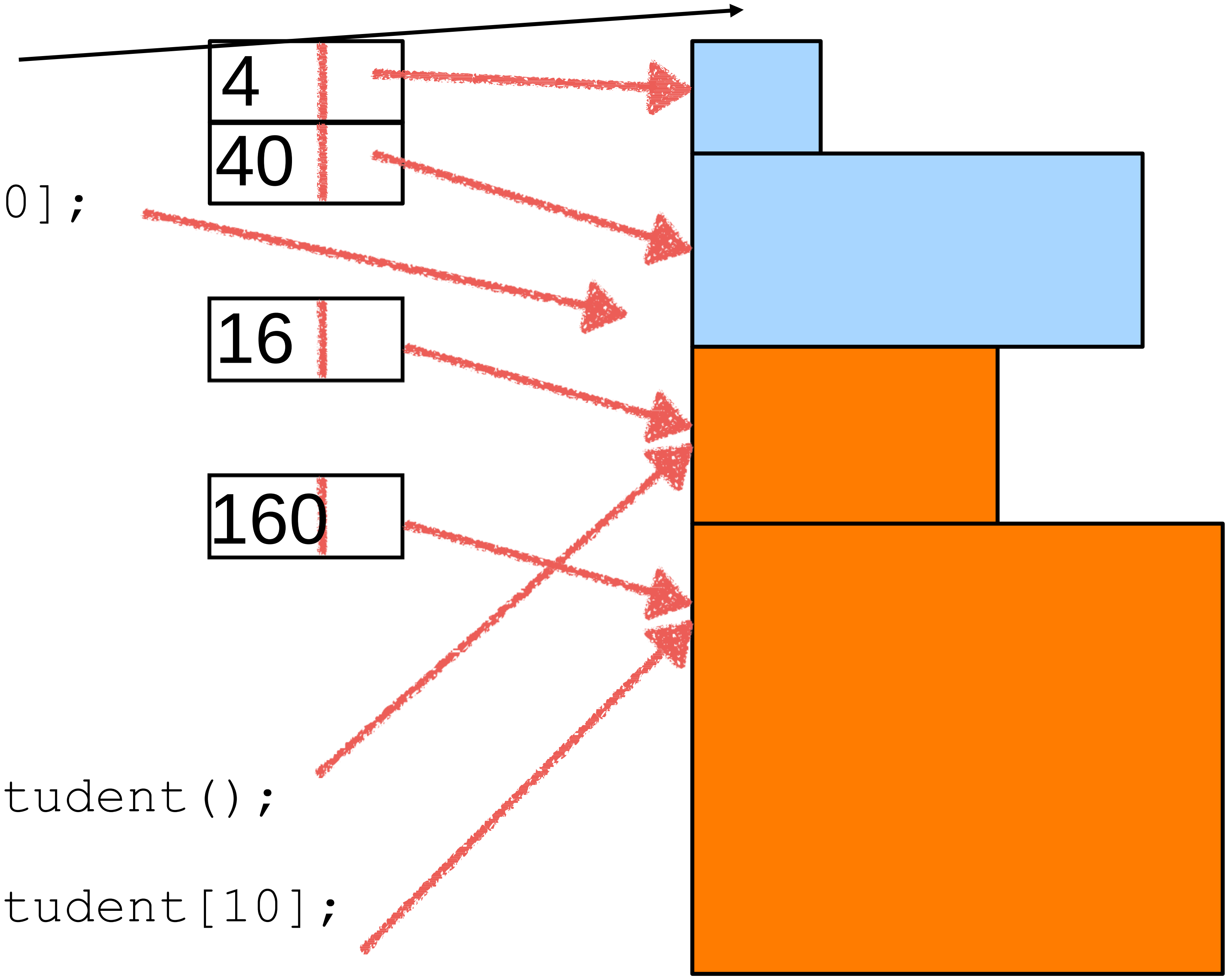


```
Student *q=new Student();

Student *r=new Student[10];
```

The new-delete Mechanism

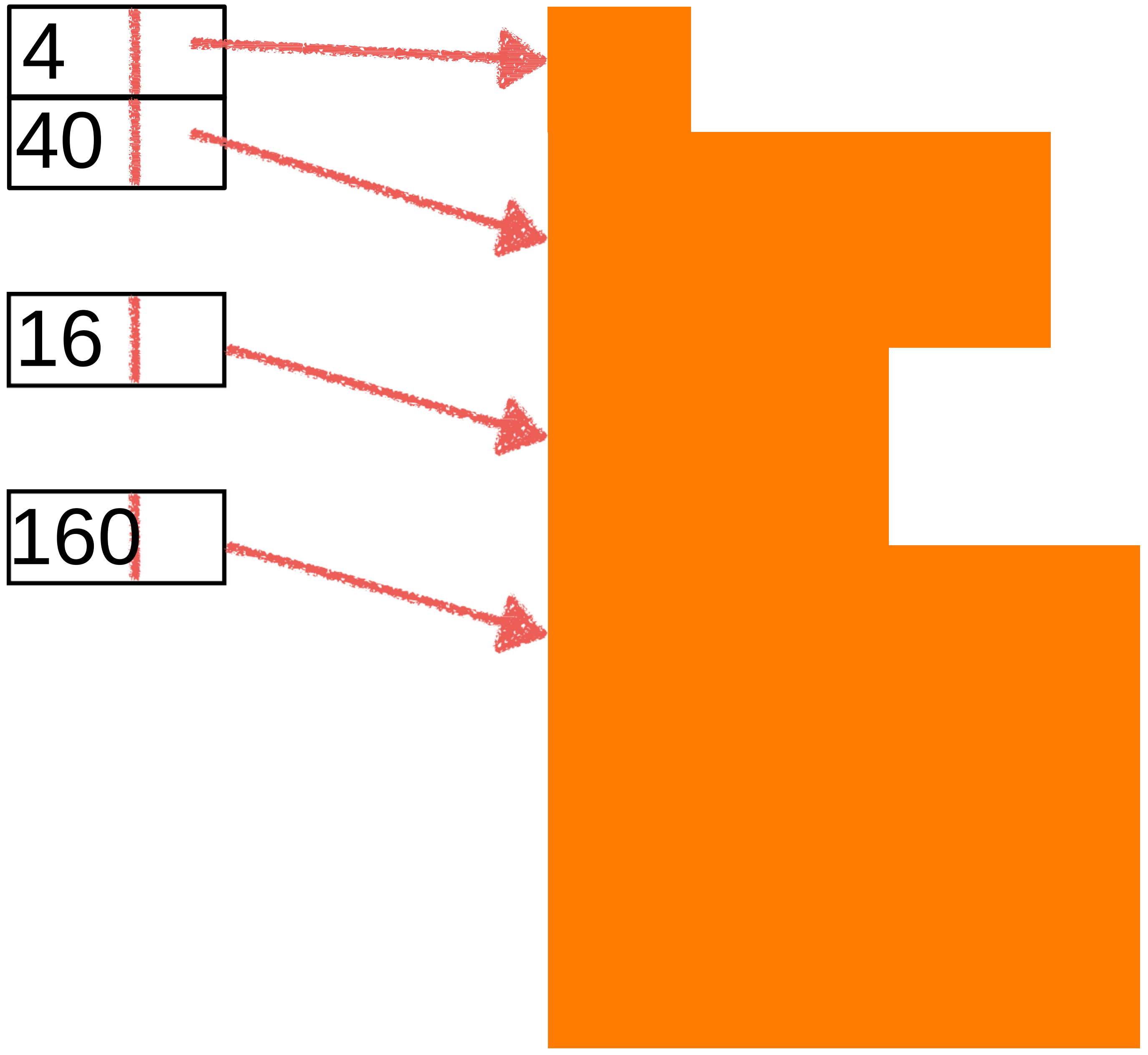
```
int *p=new int;

int *a=new int[10];
```

```
Student *q=new Student();

Student *r=new Student[10];
```



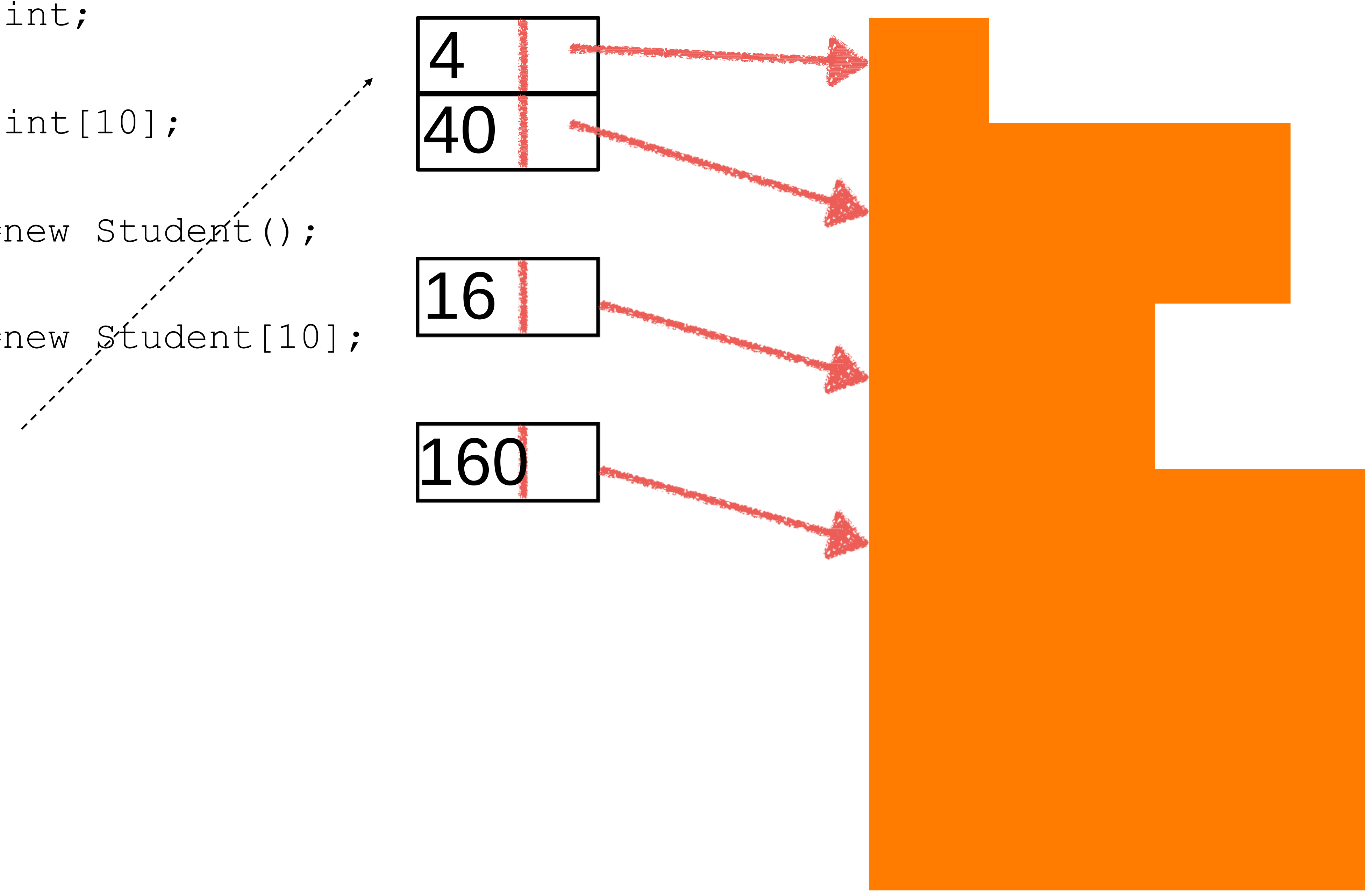
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];

Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



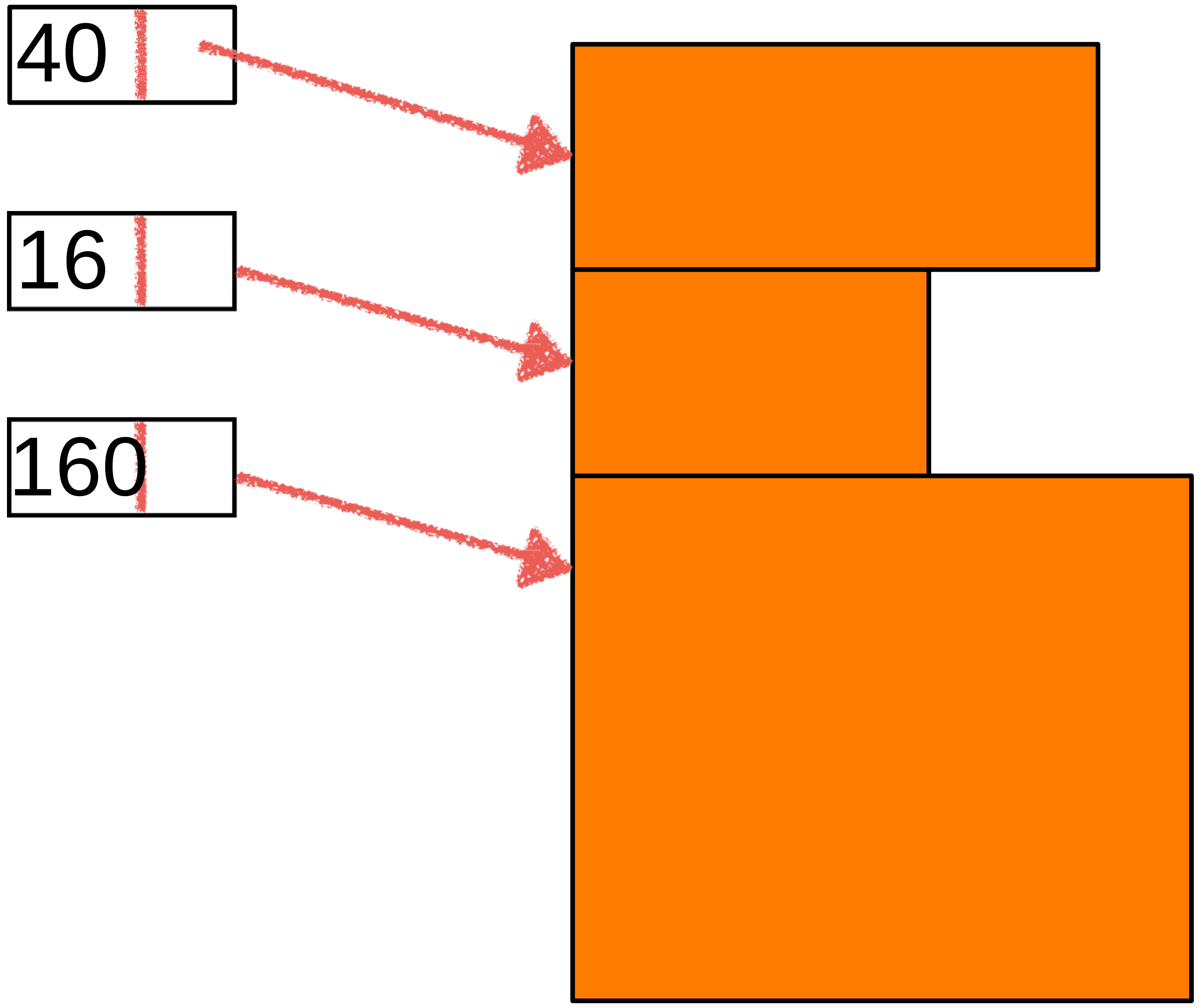
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];

Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



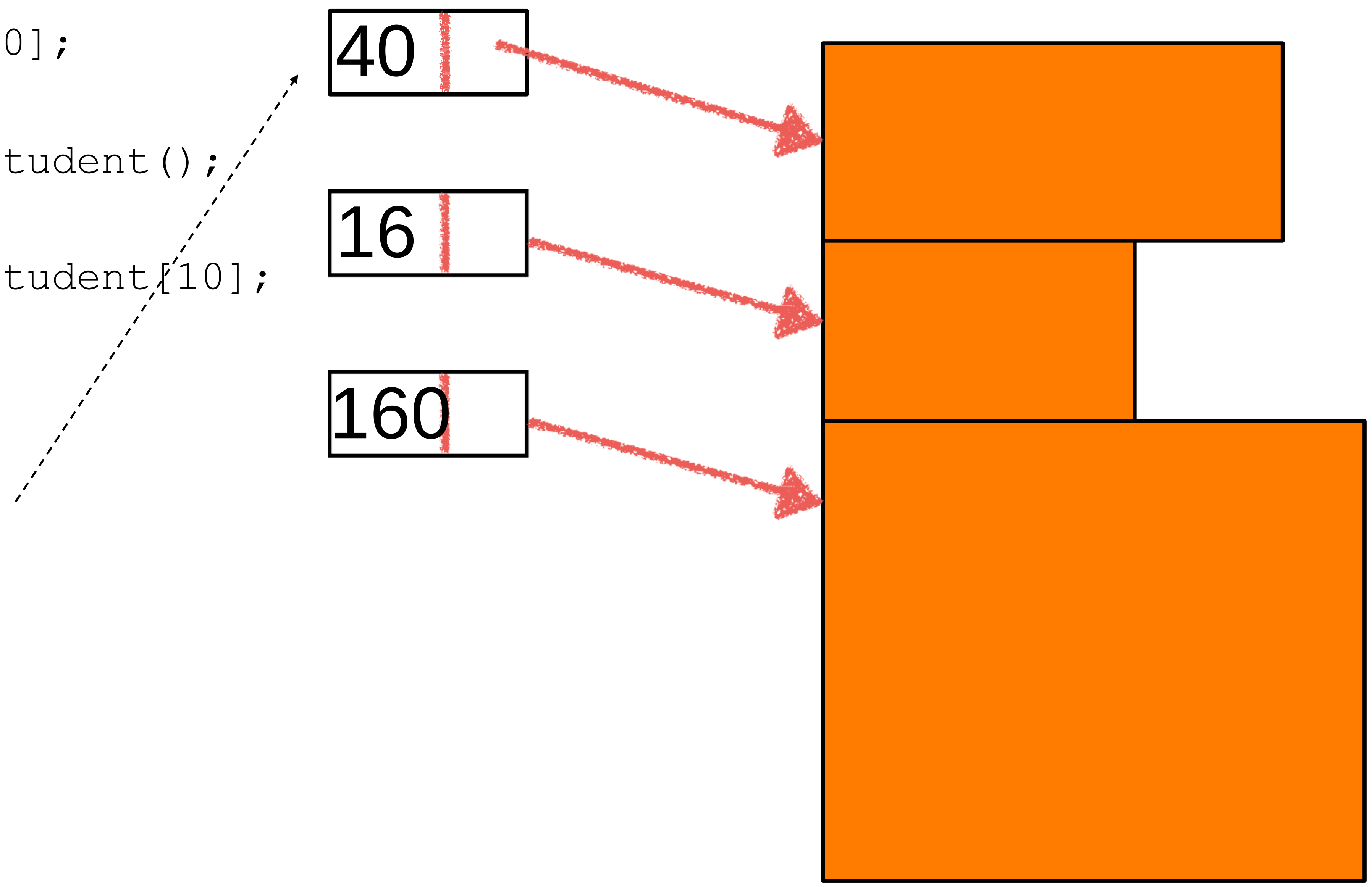
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];

Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



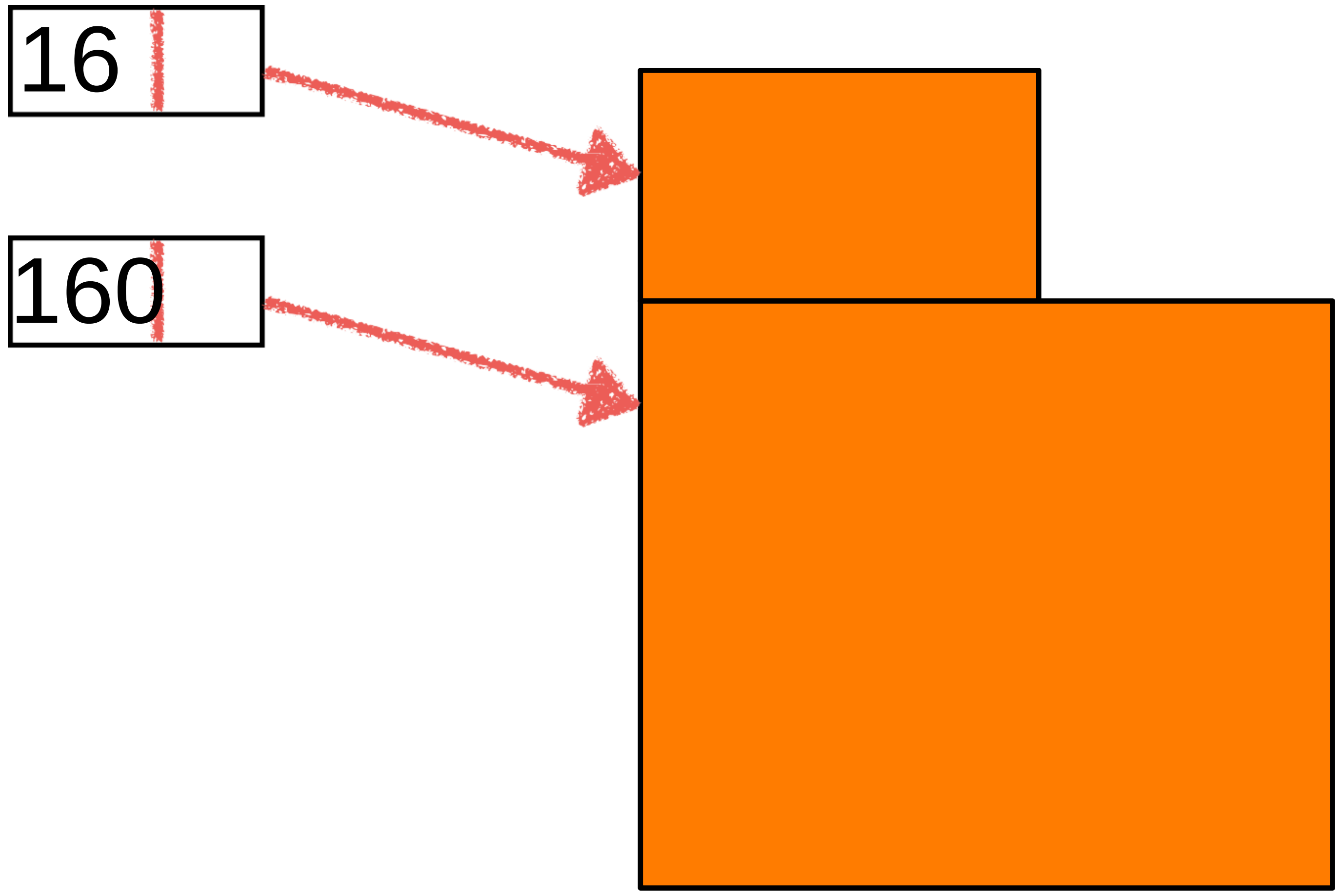
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];

Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



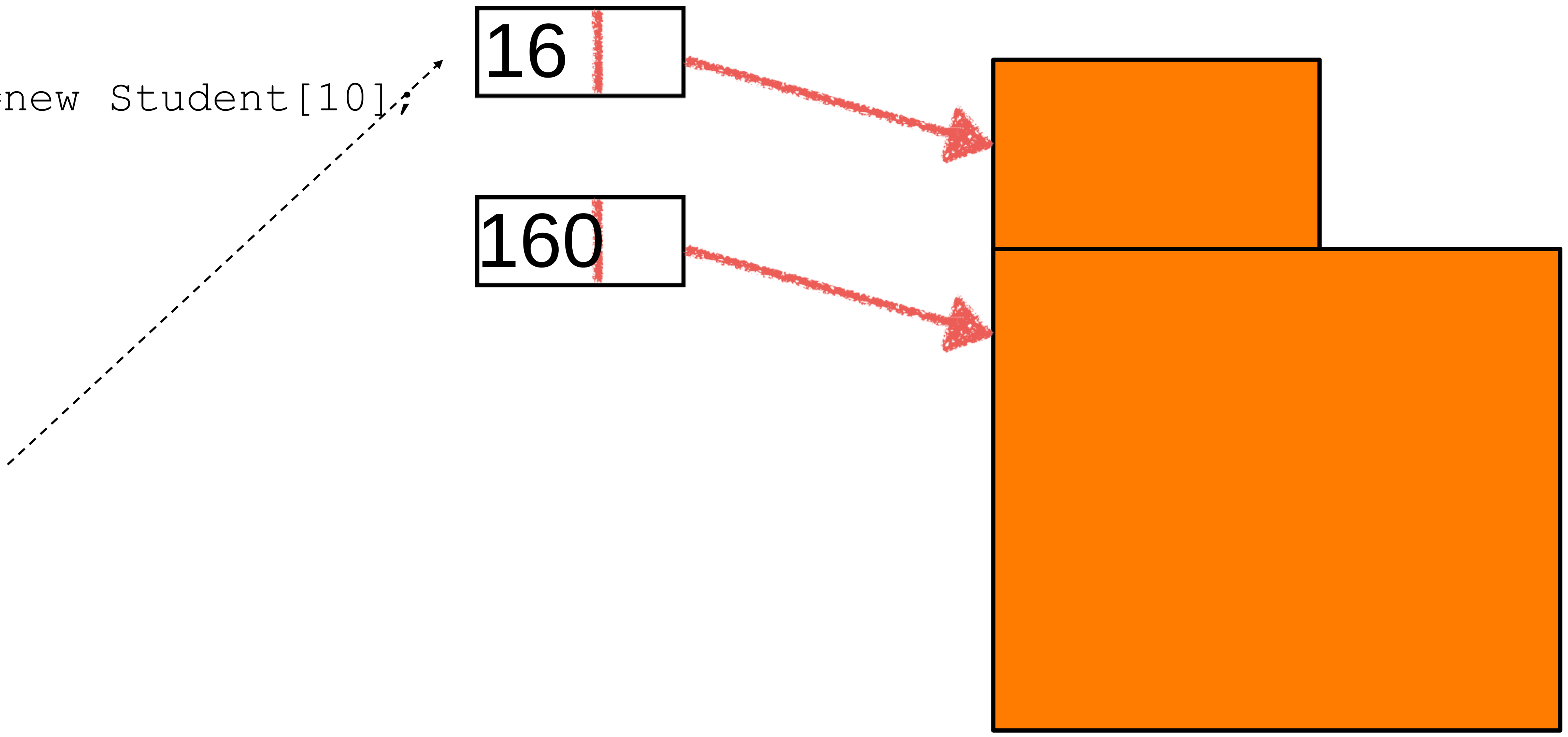
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



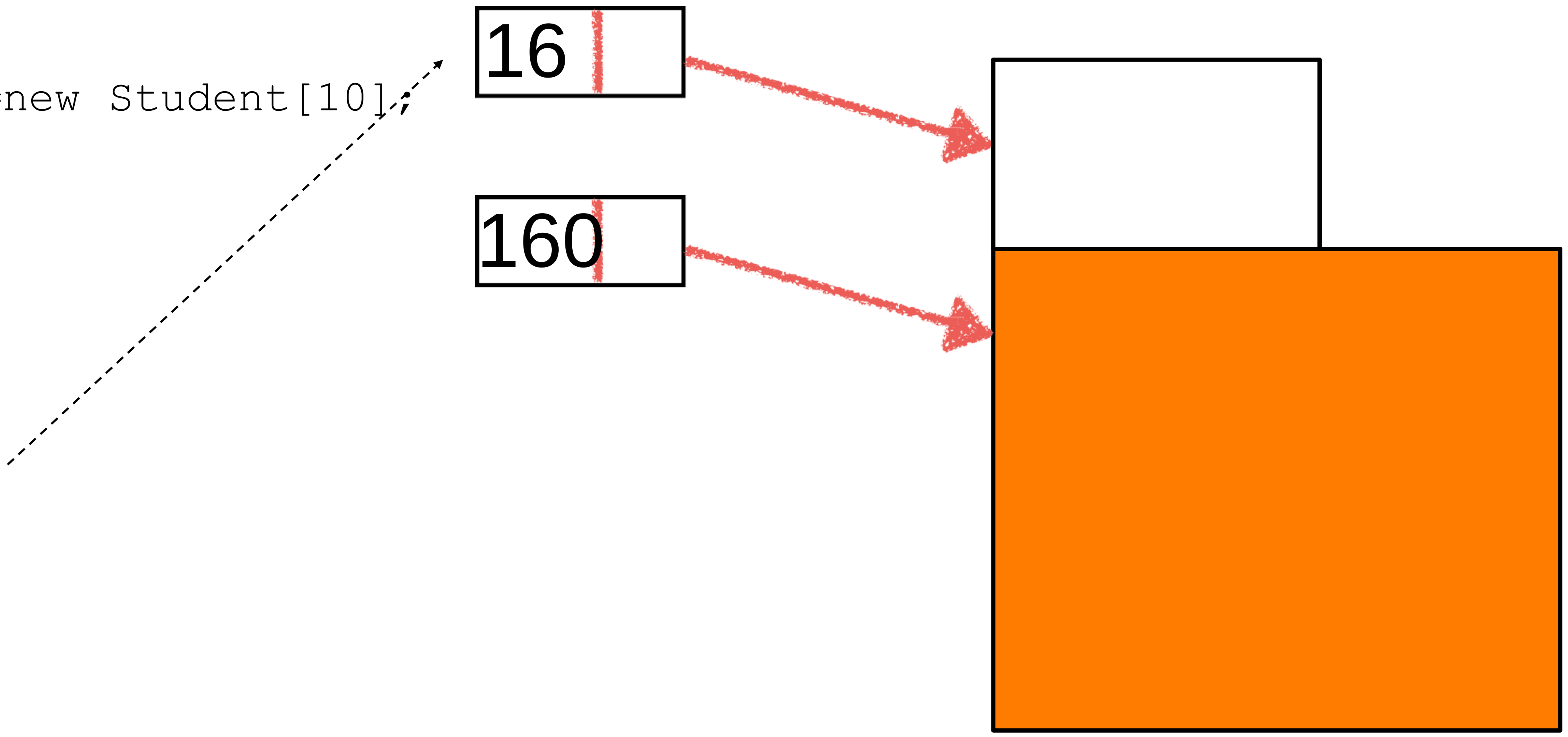
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



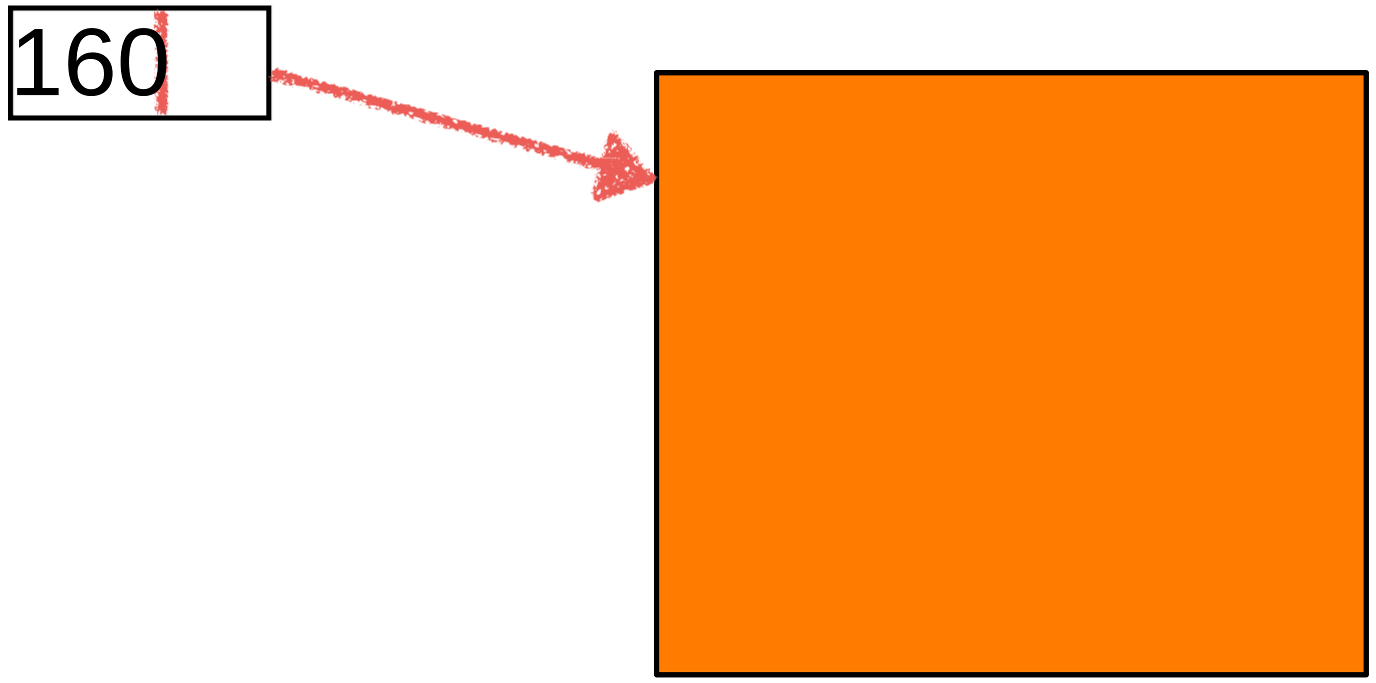
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



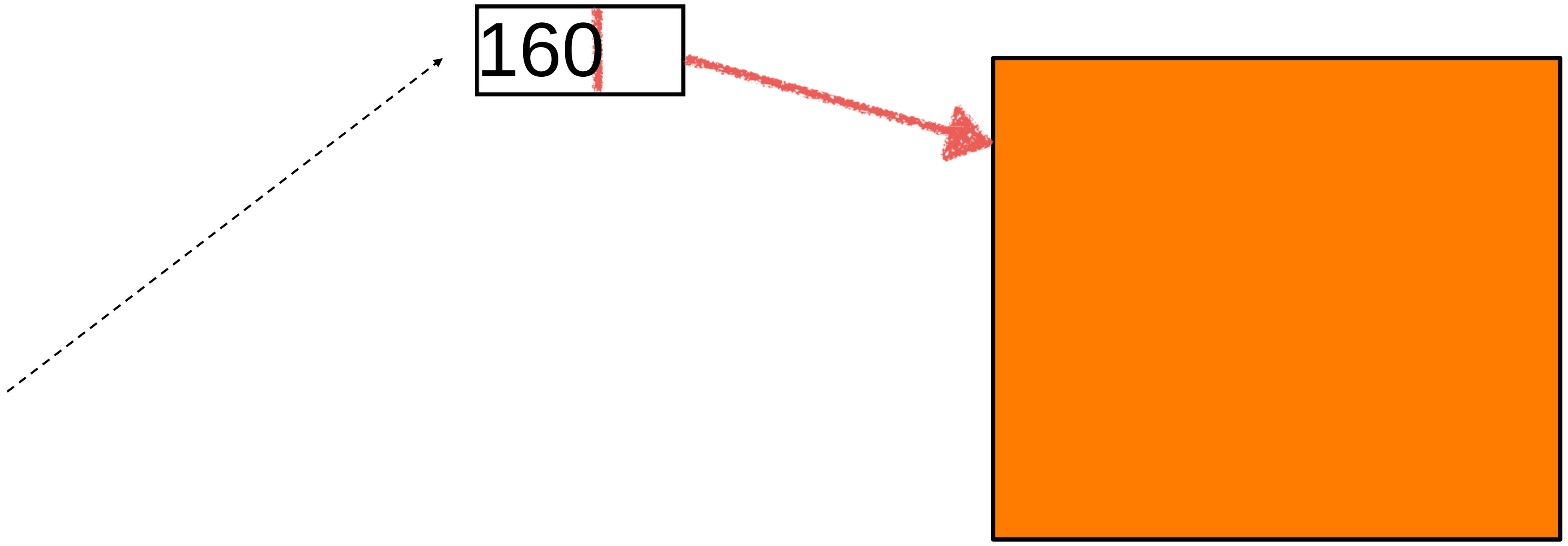
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();


Student *r=new Student[10];


delete p;


delete[] a;


delete q;


delete r;


delete[] r;
```



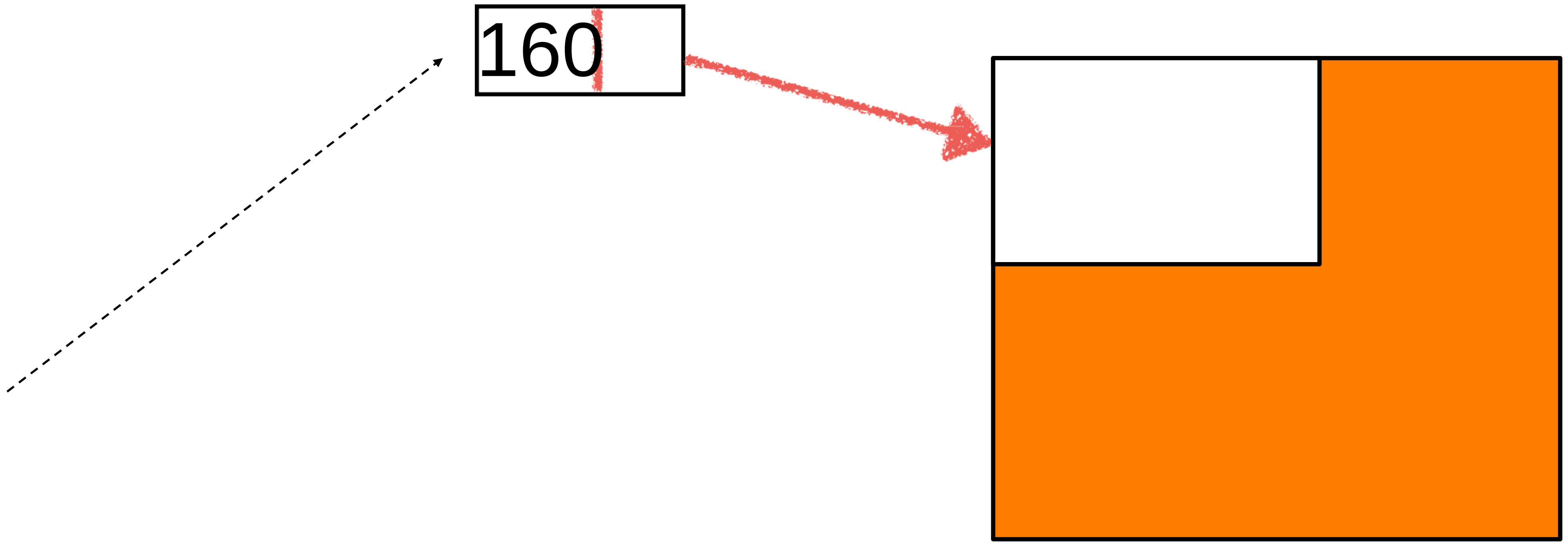
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```

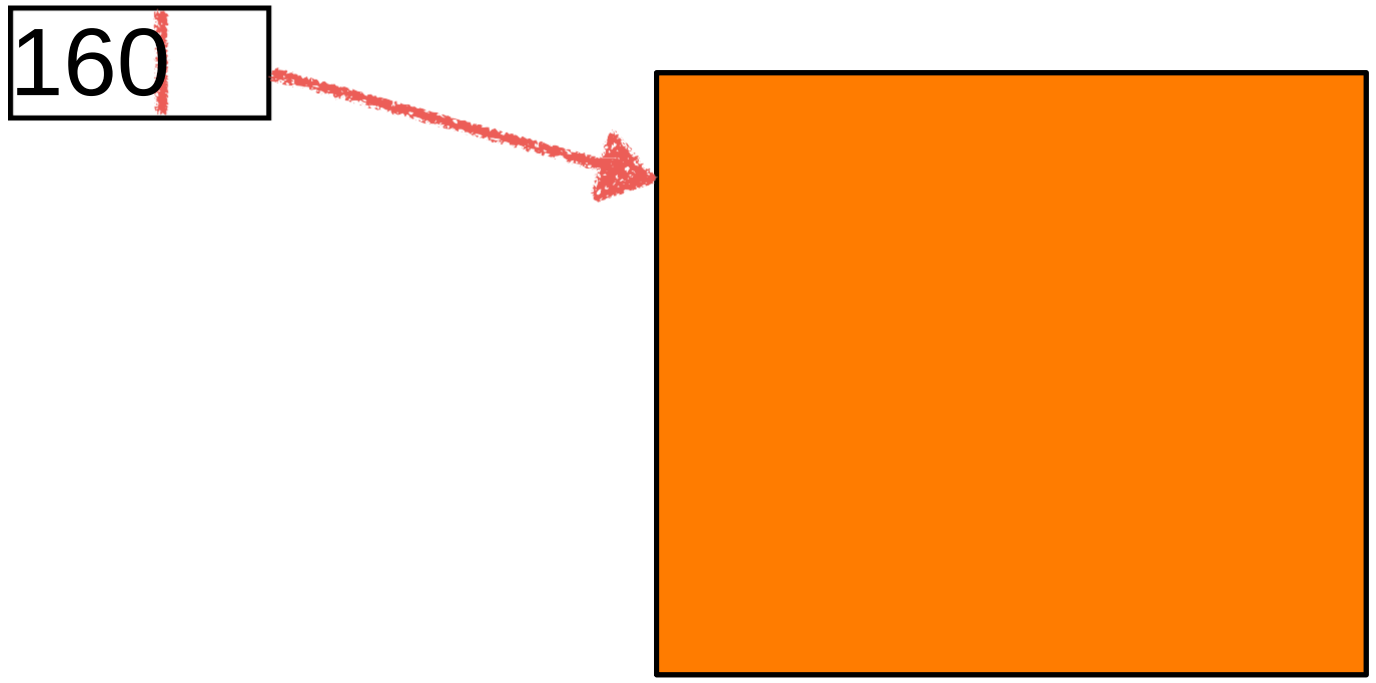


The new-delete Mechanism

```
int *p=new int;
int *a=new int[10];

Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```

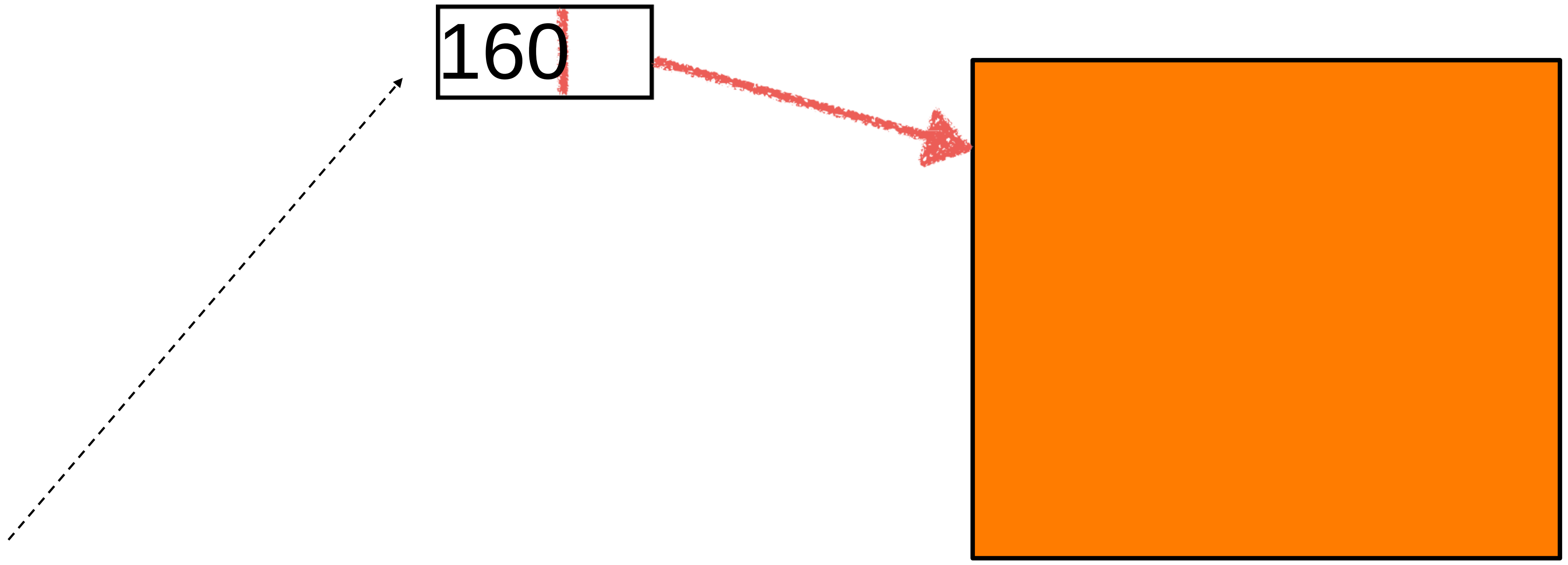
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();


Student *r=new Student[10];


delete p;


delete[] a;


delete q;


delete r;


delete[] r;
```



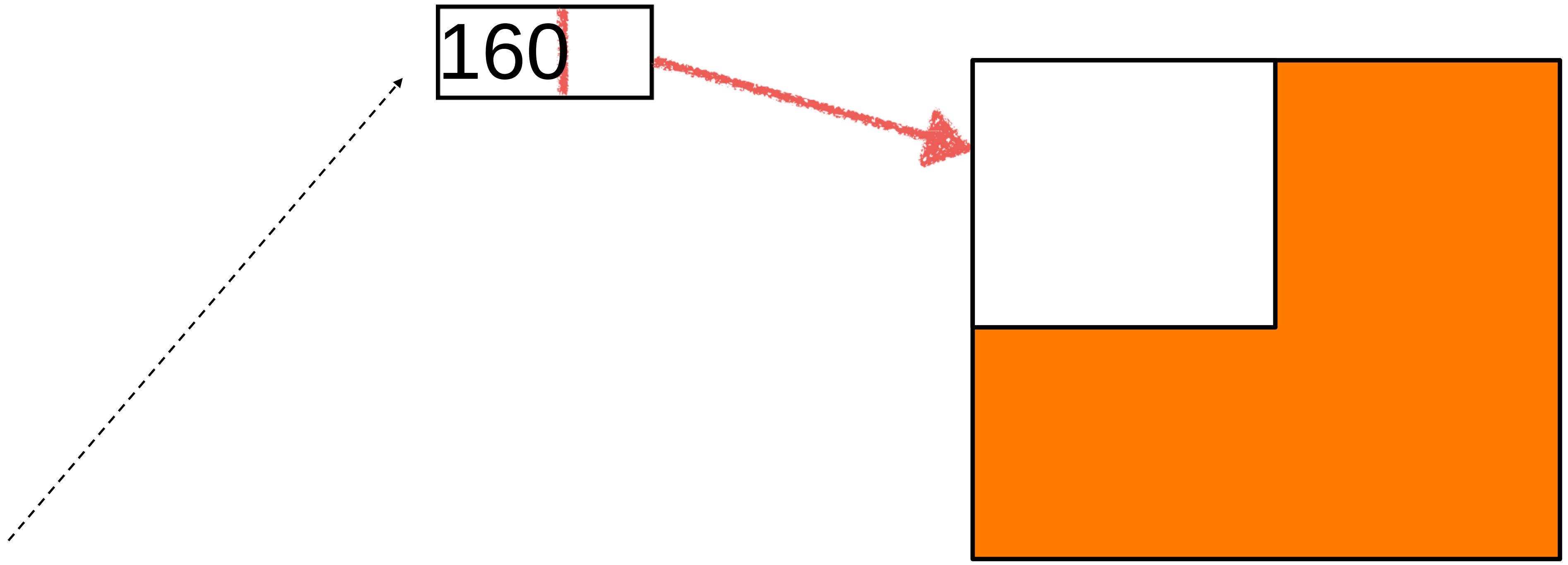
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



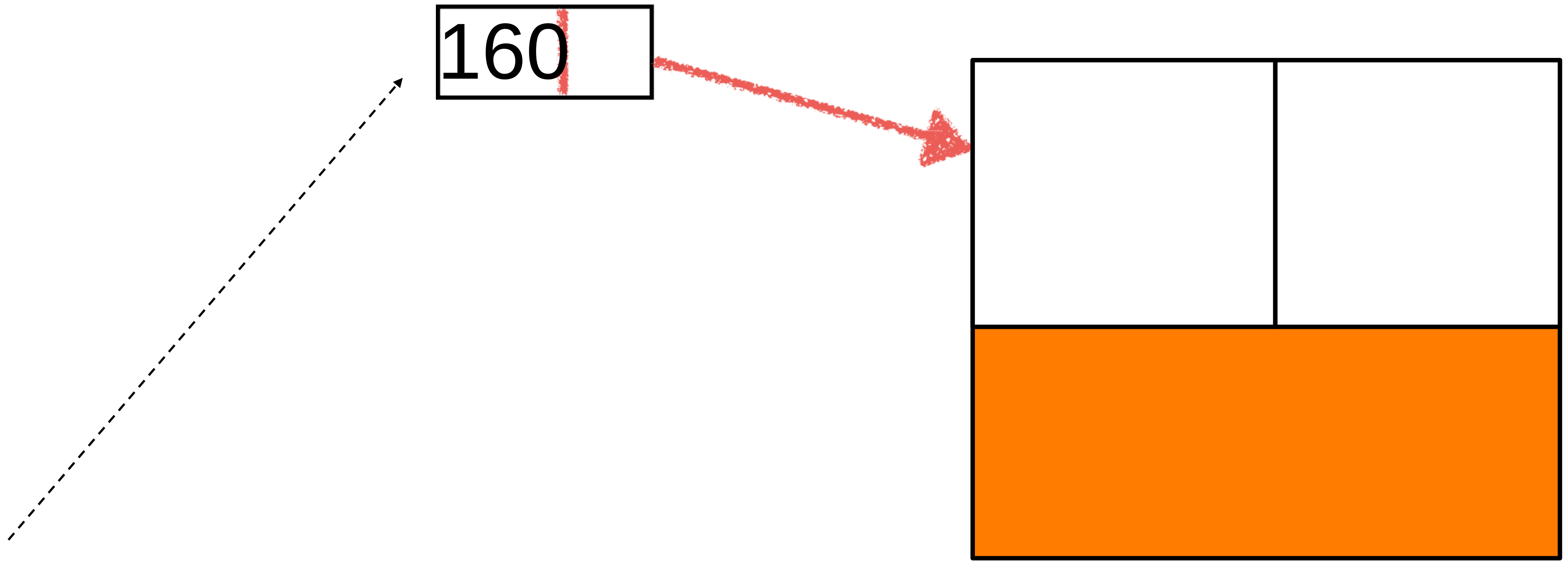
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



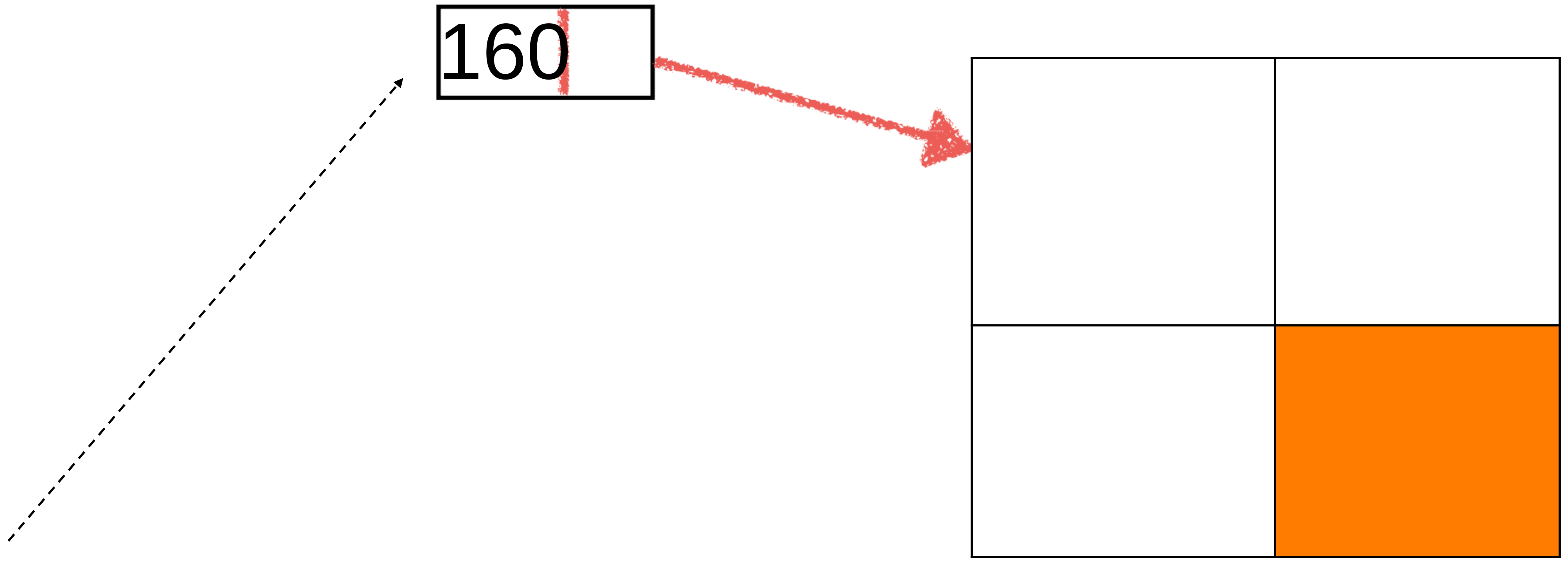
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



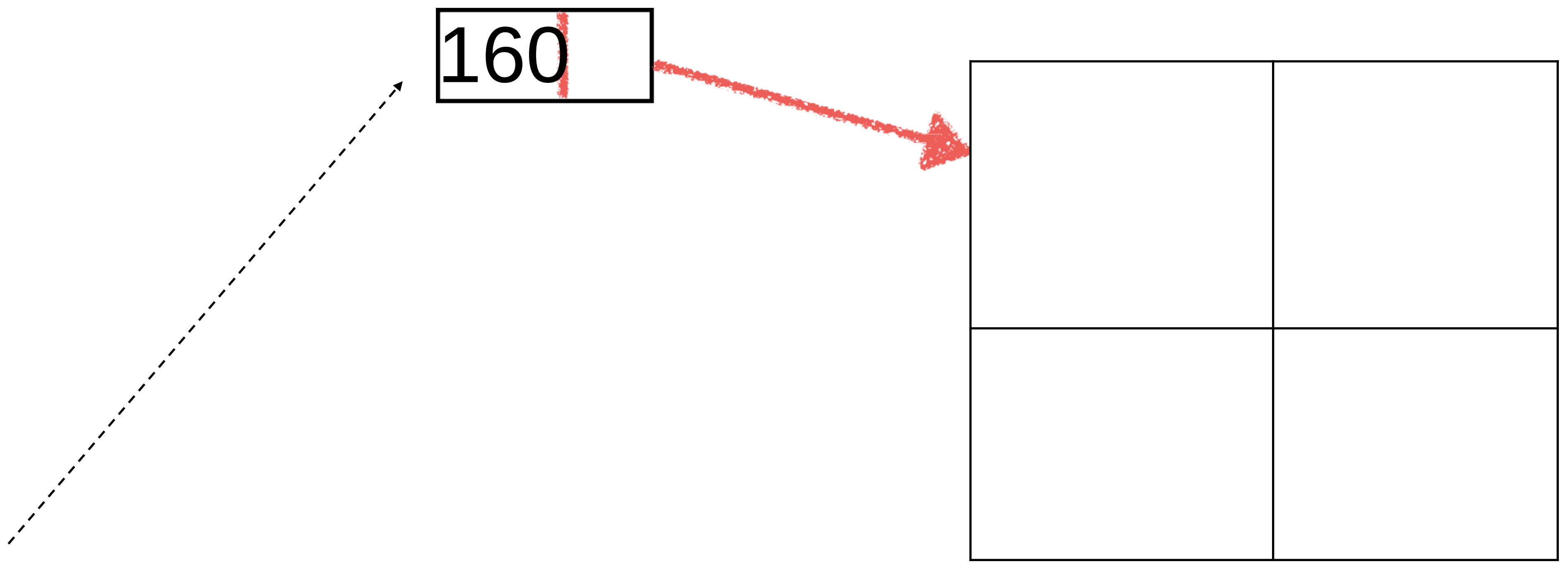
The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();


Student *r=new Student[10];


delete p;


delete[] a;


delete q;


delete r;


delete[] r;
```



The new-delete Mechanism

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();

Student *r=new Student[10];

delete p;

delete[] a;

delete q;

delete r;

delete[] r;
```



Tips for new and delete

- Don't use delete to free memory that new didn't allocate.
- Don't use delete to free the same block of memory twice in succession.
- Use delete [] if you used new [] to allocate an array.
- Use delete (no brackets) if you used new to allocate a single entity.
- It's safe to apply delete to the null pointer (nothing happens).
- Don't mix-use new/delete and malloc/free.

constant

const

- declares a variable to have a constant value

```
const int x = 123;
x = 27; // illegal!
x++; // illegal!

int y = x; // ok, copy const to non-const
y = x; // ok, same thing
const int z = y; // ok, const is safer
```

Constants

- Constants are like variables
  - Observe scoping rules
  - Declared with “const” type modifier

Constants

- Constants are like variables
  - Observe scoping rules
  - Declared with "const" type modifier
- A const in C++ defaults to *internal linkage*
  - the compiler tries to avoid creating storage for a const -- holds the value in its symbol table.
  - extern forces storage to be allocated.

Compile time constants

- Compile time constants are entries in compiler symbol table, not really variables.

```
const int bufsize = 1024;
```

- Value must be initialized
- unless you make an explicit extern declaration:

```
extern const int bufsize;
```

- Compiler won't let you change it

Run-time constants

- const value can be exploited

```
const int class_size = 12;
int finalGrade[class_size]; // ok
int x;
cin >> x;
const int size = x;
double classAverage[size]; // error
```

Aggregates

- It’s possible to use **const** for aggregates, but storage will be allocated. In these situations, **const** means “a piece of storage that cannot be changed.” However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time.

```
const int i[] = { 1, 2, 3, 4 };
float f[i[3]]; // Illegal

struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };

double d[s[1].j]; // Illegal
```

Pointers with const

P: 0xaffefado ==> a: [53,54,55]

```
int a[] = {53,54,55};
int * const p = a; // p is const
*p = 20; // OK
p++; // ERROR

const int *p = a; // (*p) is const
*p = 20; // ERROR!
p++; // OK
```

Quiz: What do these mean?

```
string s( "Fred" );
const string*   p   =   &s;
string  const* p    =   &s;
string  *const p    =   &s;
```

Pointers and constants

	`int i;`	`const int ci = 3;`
`int * ip;`	`ip = &i;`	`ip = &ci; //Error`
`const int *cip`	`cip = &i;`	`cip = &ci;`

Remember:

```
*ip  = 54;  // always legal since ip points to int
*cip = 54;  // never legal since cip points to const int
```

String Literals

```
char* s = "Hello, world!";
```

- s is a pointer initialized to point to a string constant
- This is actually a `const char* s` but compiler accepts it without the const
- Don't try and change the character values (it is undefined behavior)

- If you want to change the string, put it in an array:

```
char s[] = "Hello, world!";
```

Conversions

- Can always treat a non-const value as const

```
void f(const int* x);
int a = 15;
f(&a); // ok
const int b = a;

f(&b); // ok
b = a + 1; // Error!
```

- You cannot treat a constant object as non-constant without an explicit cast (const_cast)

Passing by const value?

- Can always treat a non-const value as const

```
void f1 (const int i) {
 i++; // illegal: compile-time error
}
```

Returning by const value?

```
int f3() { return 1; }
const int f4() { return 1; }

int main() {
    const int j = f3(); // works fine
    int k = f4(); // this works fine too
}
```

Passing addresses

- Passing large objects are expensive.
- Better to pass by address, using a pointer or a reference.
- Make it const whenever possible to prevent unexpected modification.