Using Objects

Object-Oriented Programming in C++

Zhaopeng Cui

Safe way to read a string in?

string

The string class

- You must add this at the head of you code
  - `#include <string>`

- Define variable of string like other types
  - `string str;`

- Initialize it w/ string contant
  - `string str = “Hello”;`

- Read and write string w/ cin/cout
  - `cin >> str;`
  - `cout << str;`

Assignment for string

```
char charr1[20];

char charr2[20] = “jaguar”;


string str1;

string str2 = “panther”;


carr1 = char2;    //  illegal

str1 = str2;   //  legal
```

Concatenation for string

- `string str3;`
- `str3 = str1 + str2;`
- `str1 += str2;`
- `str1 += “lalala”;`

ctors

- string(const char *cp, int len);
- string(const string& s2, int pos);
- string(const string& s2, int pos, int len);

sub-string

- substr(int pos, int len);

alter string

- assign();
- insert(const string&, int len);
- insert(int pos, const string& s);
- erase();
- append();
- replace();

search string

- find()

File I/O

- #include <fstream>

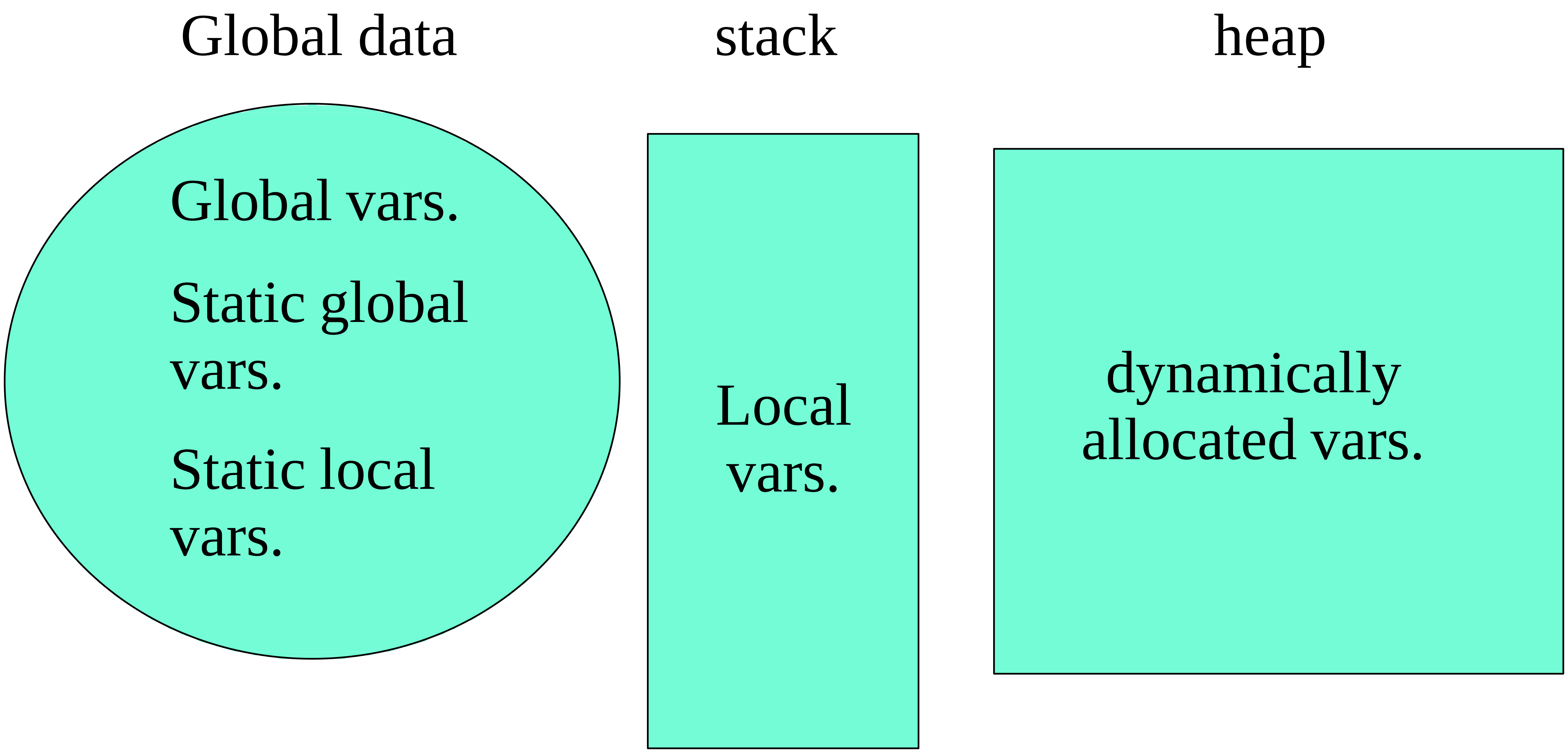
```
ofstream File1(“test.txt”);

File1<<“Hello world”<<std::endl;

ifstream File2(“test.txt”);

std::string str;

File2>>str;
```

Memory Model

What are they?

```
int i;    // global vars.

string  str;

static  int j;  //static global vars.

f() {
    int k;     // local vars.

    static int l;  // static local

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

Where are they?



global vars

- vars defined outside any functions
- can be shared btw .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 btw visit to the 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
- string *ps;
  - ps is a pointer to an object

- string s;
  - At this line, object s is created and initialized
- string *ps;
  - At this line, the object ps points to is not known yet.

Assignment

- string s1, s2;
  - s1 = s2;
- string *ps1, *ps2;
  - ps1 = ps2;

dynamically allocated memory

Dynamic memory allocation

- **new**
  - new int;
  - new Stash;
  - new int[10];
- **delete**
  - `delete p;`
  - `delete[] p;`

new and delete

- new is the way to allocate memory as a program runs. Pointers become the only access to that memory
- delete enables you to return memory to the memory pool when you are finished with it.

Dynamic Arrays

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

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

```
delete [] psome;
```

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

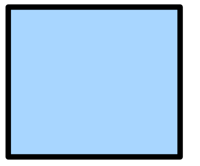
The new-delete mech.

```
int *p=new int;

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

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

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

The new-delete mech.

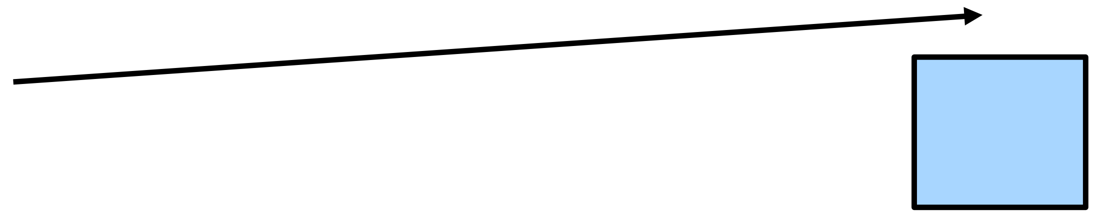
```
int *p=new int;

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



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

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

The new-delete mech.

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

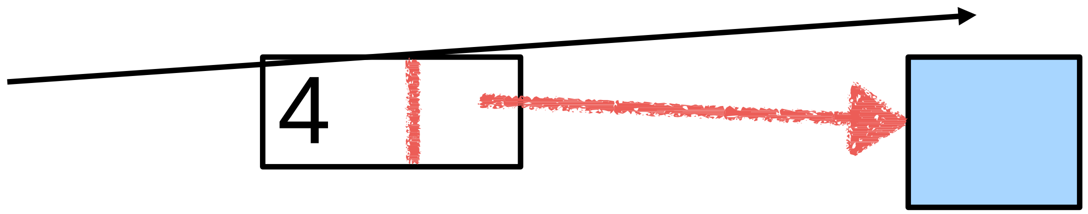


```
int *a=new int[10];
```

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

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

The new-delete mech.

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

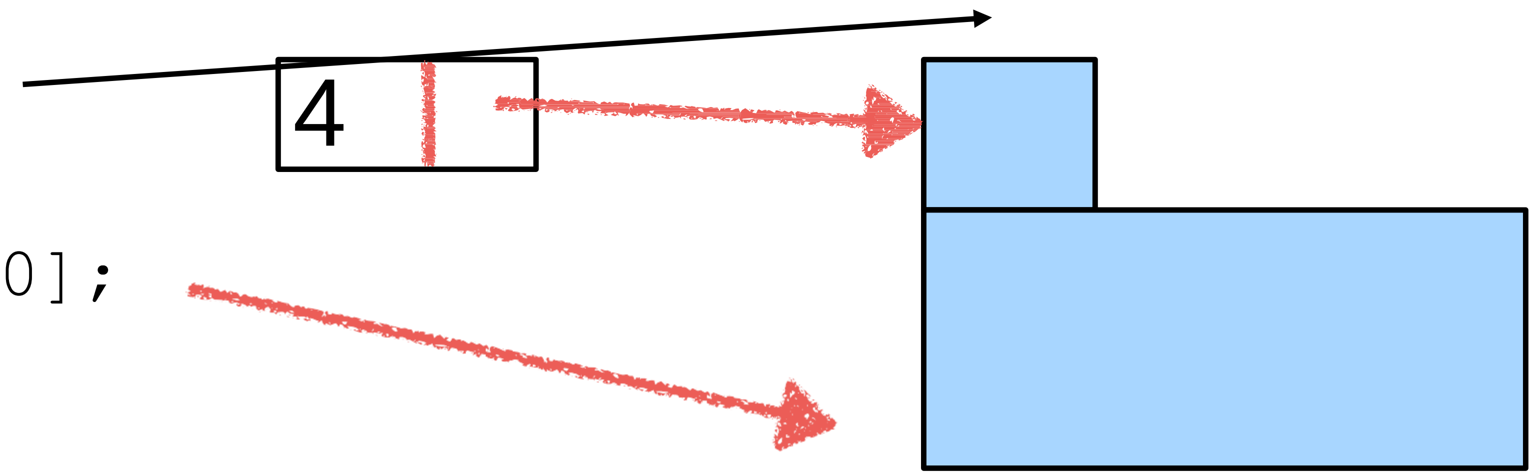



```
int *a=new int[10];
```

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

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

The new-delete mech.

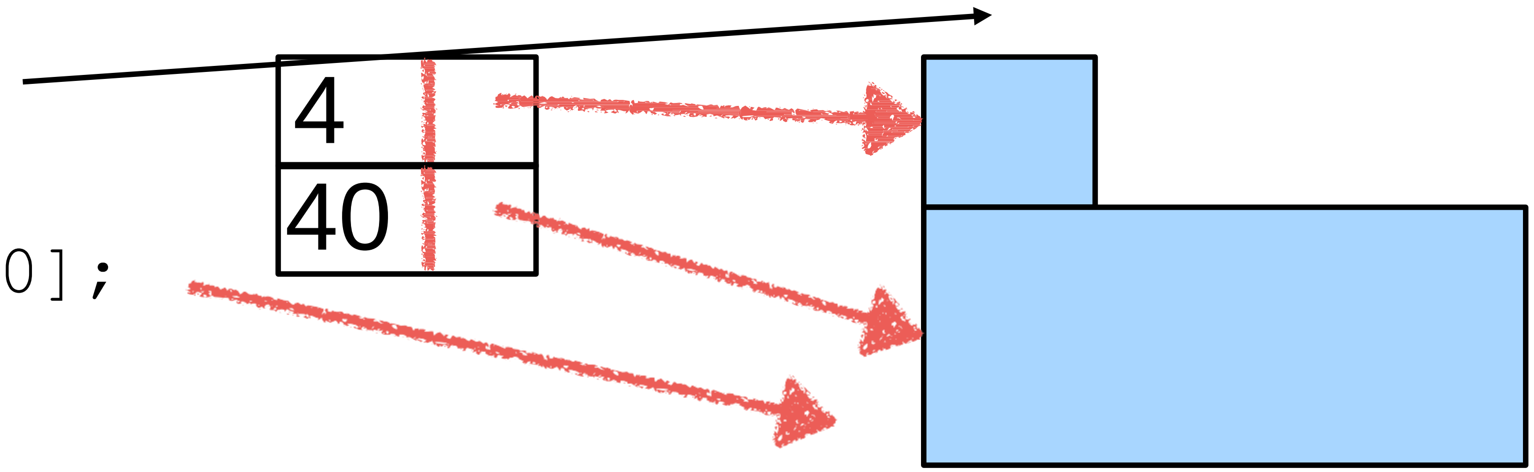
```
int *p=new int;

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




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

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

The new-delete mech.

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

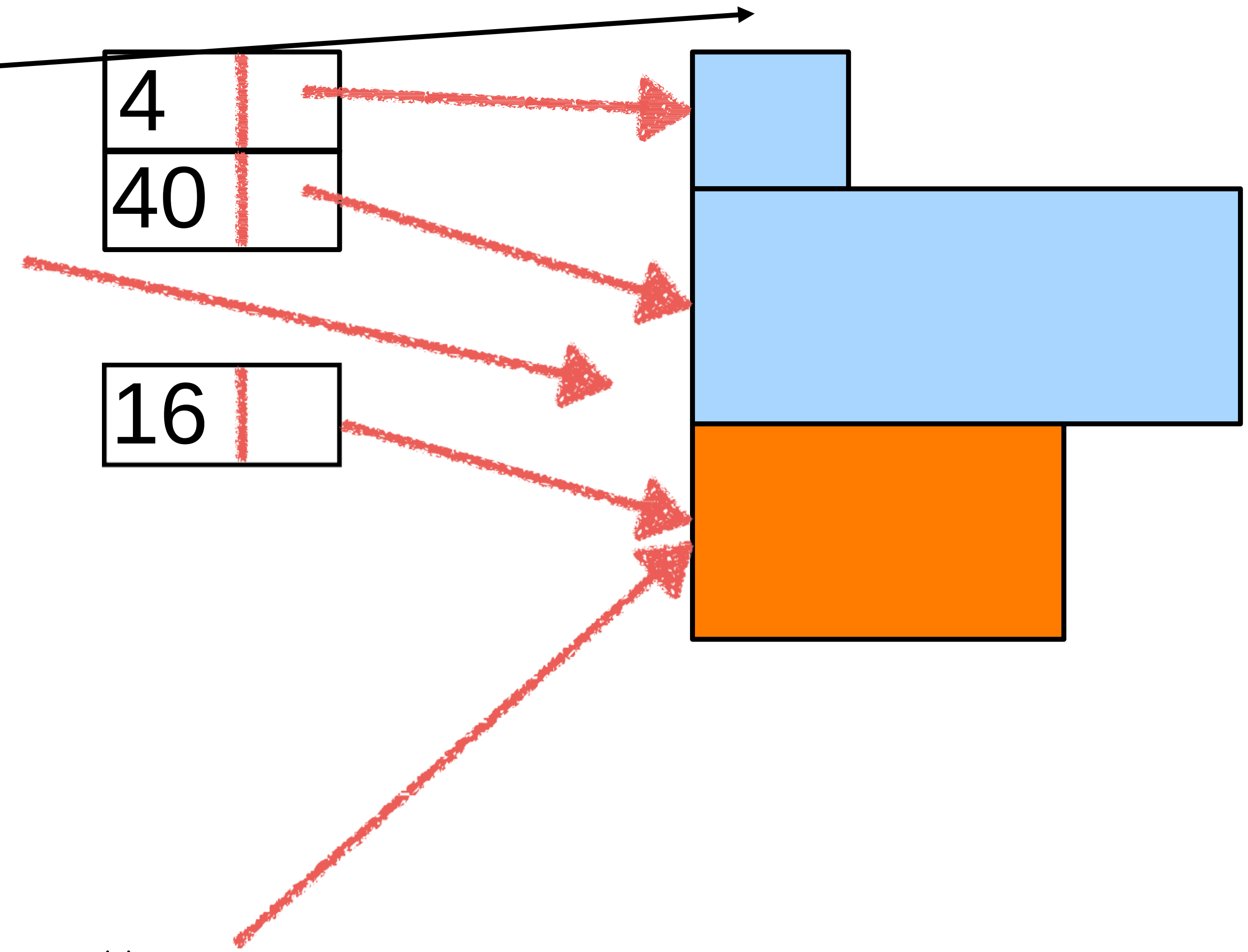



```
int *a=new int[10];
```

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

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

The new-delete mech.

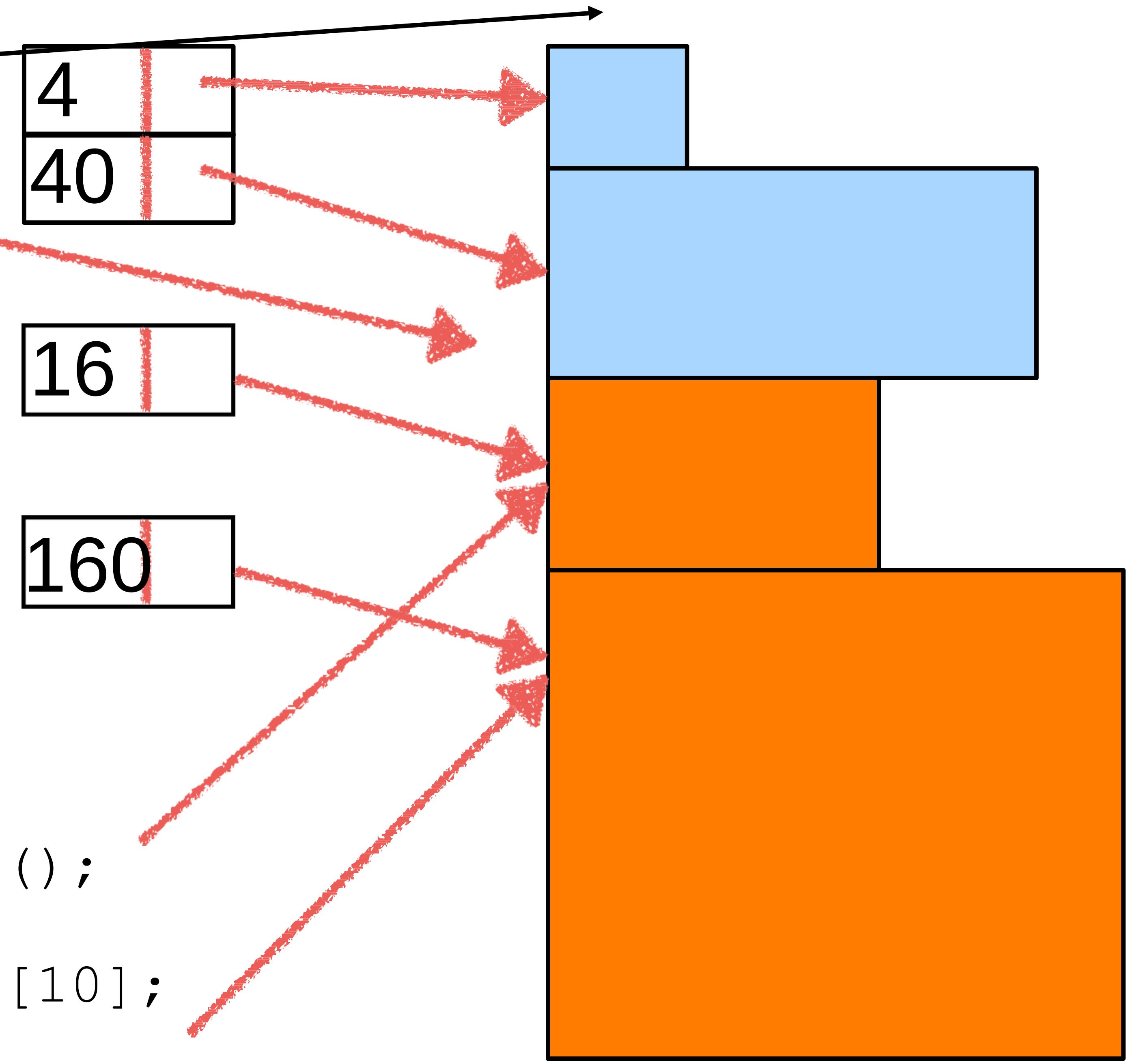
```
int *p=new int;

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




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

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

The new-delete mech.

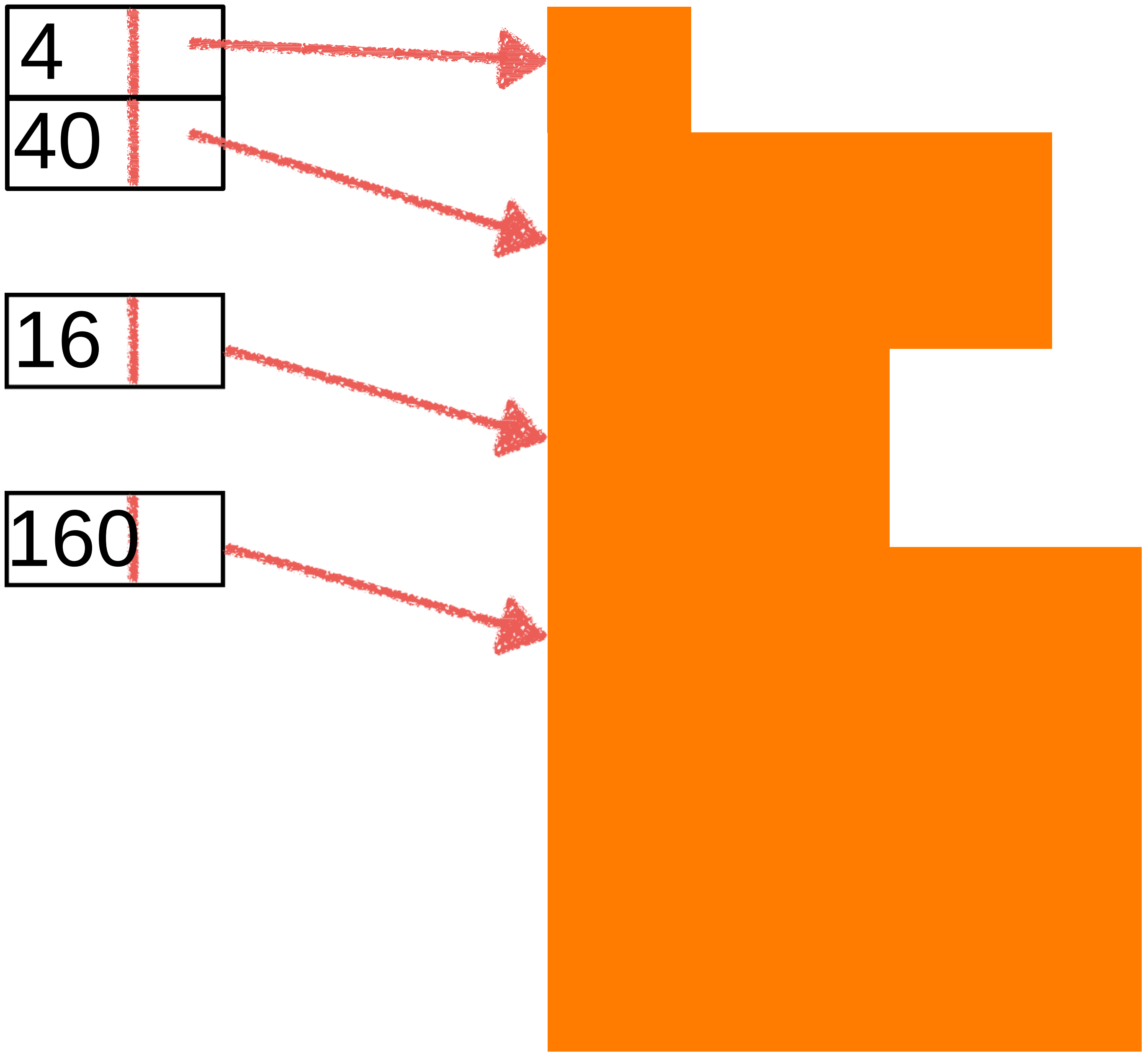
```
int *p=new int;

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




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

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

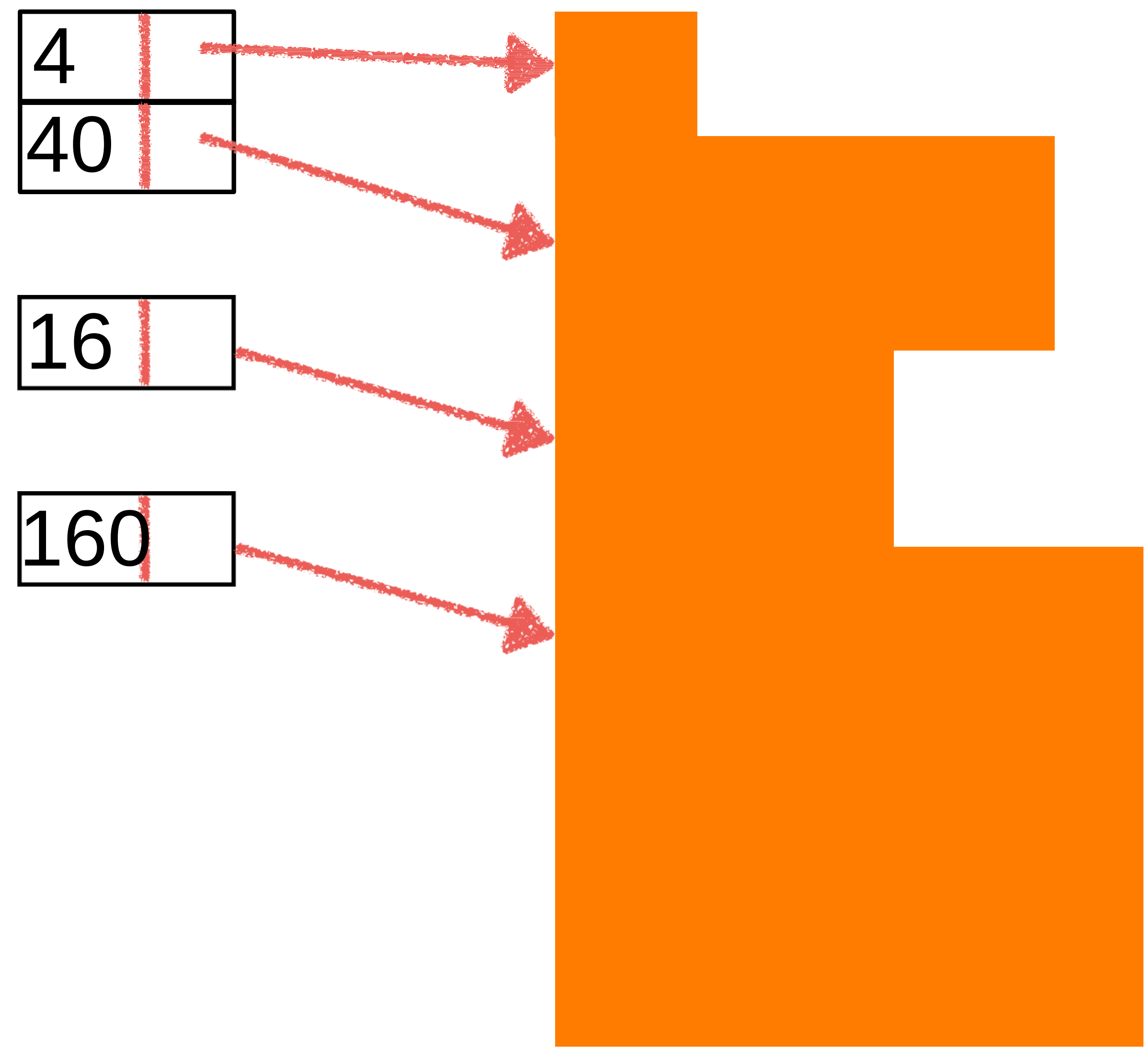
The new-delete mech.

```
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;
```



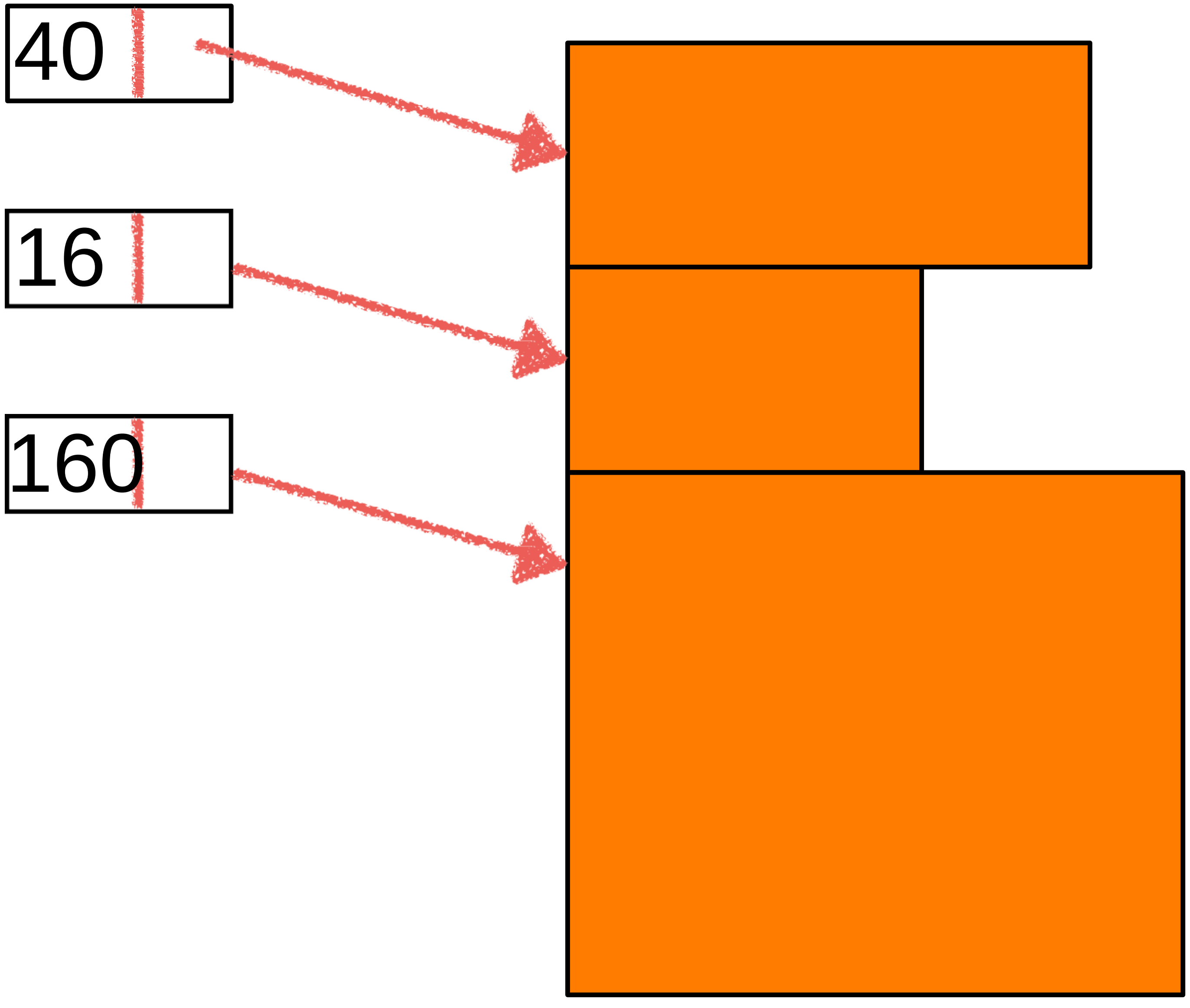
The new-delete mech.

```
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;
```



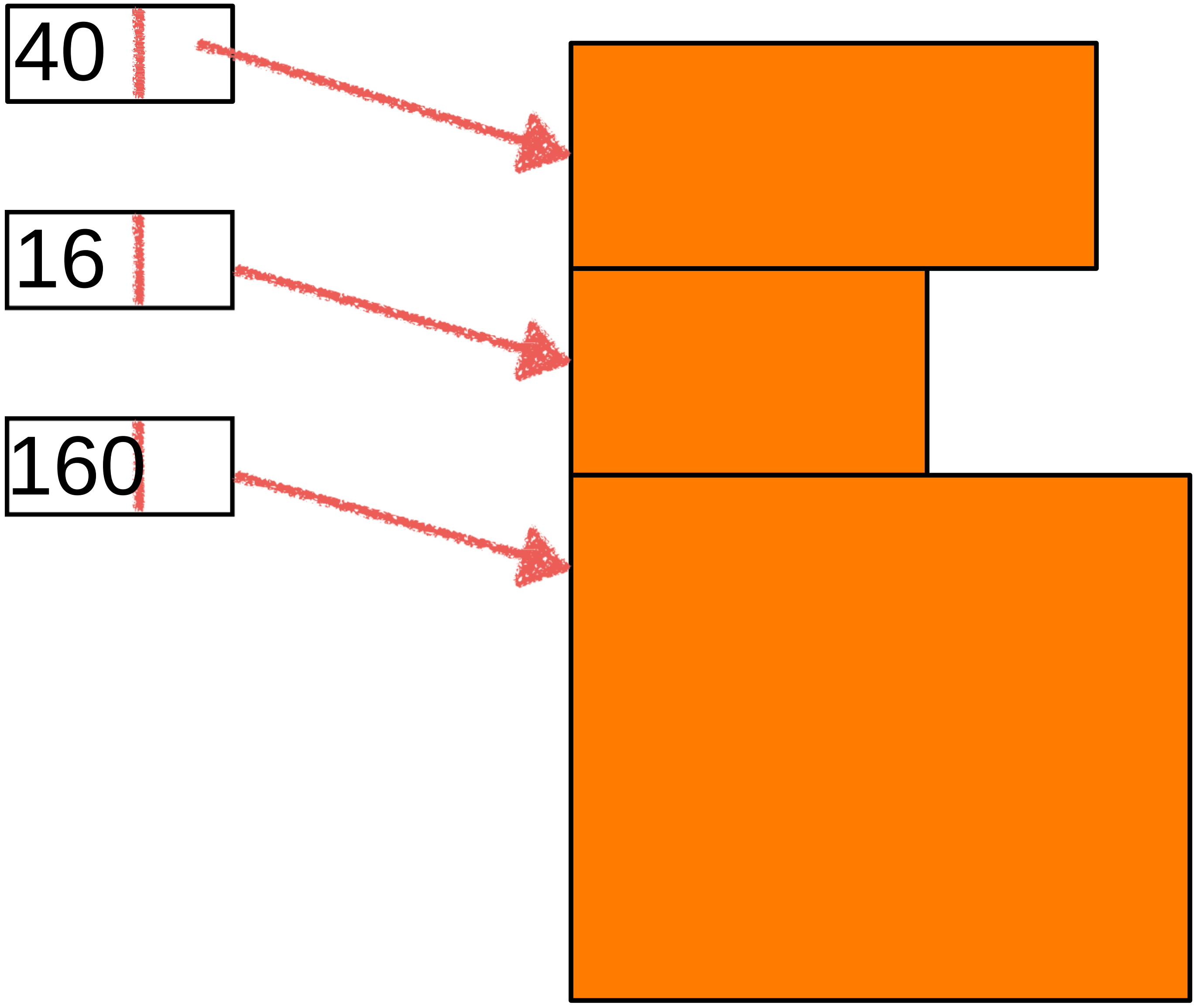
The new-delete mech.

```
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;
```



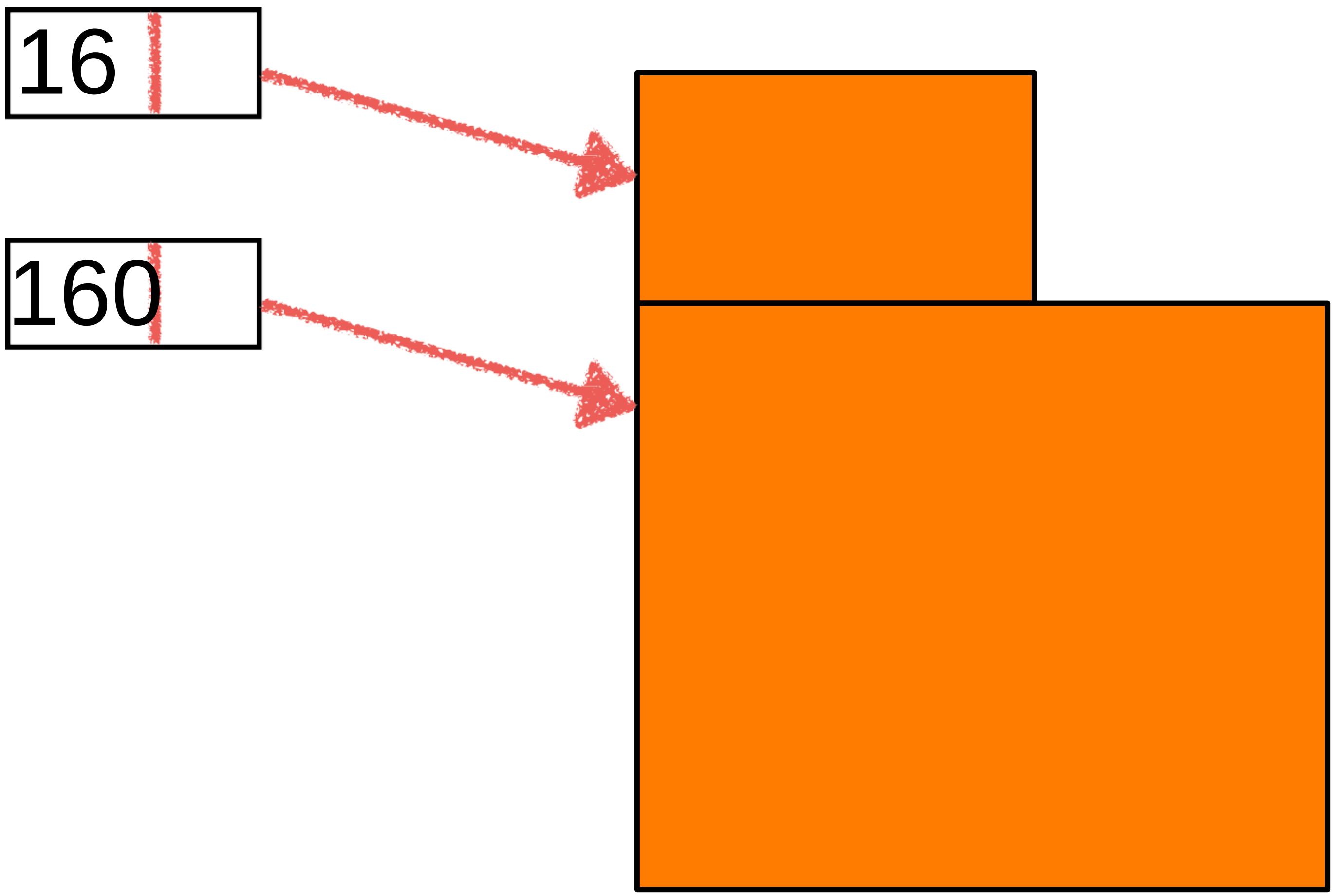
The new-delete mech.

```
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;
```



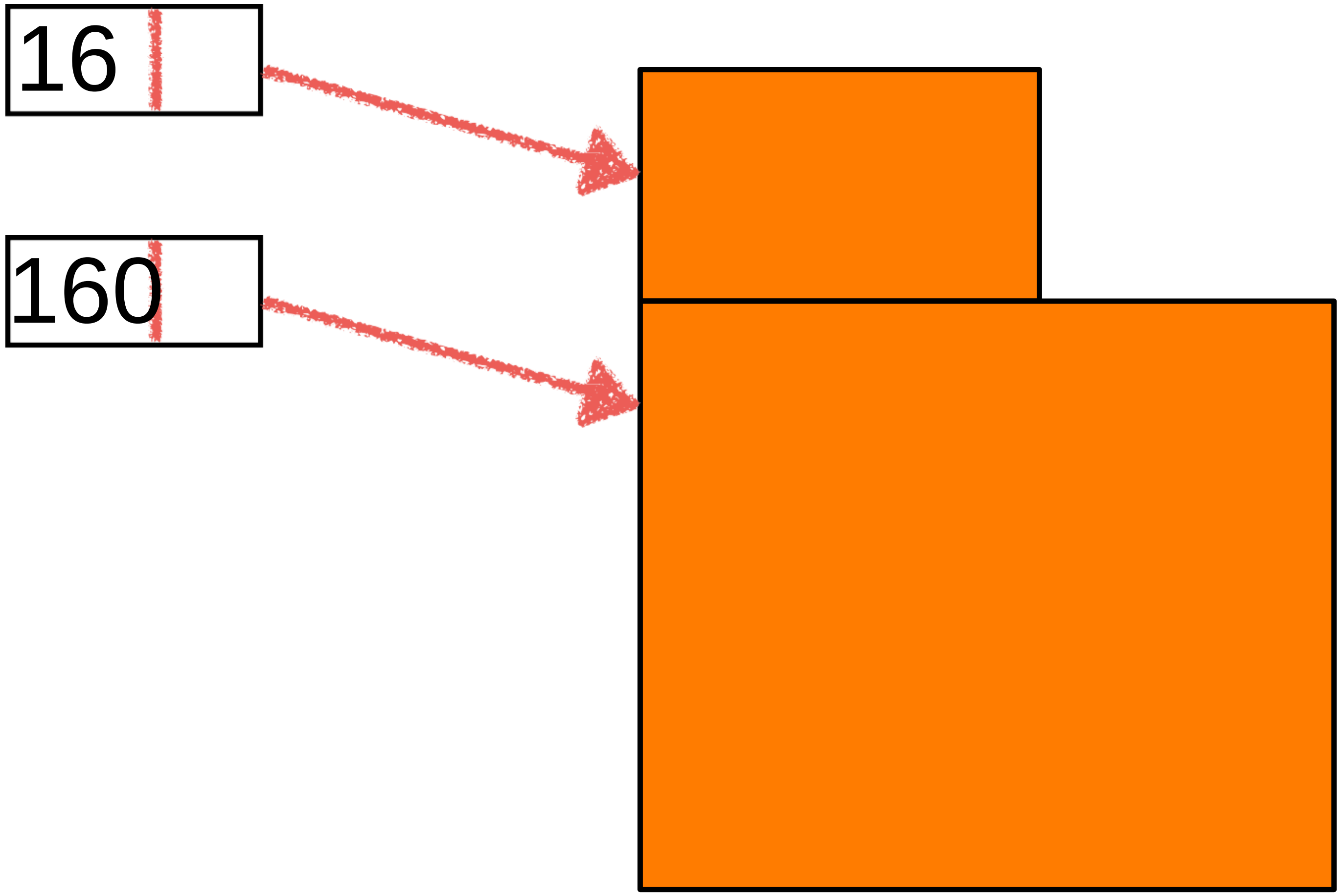
The new-delete mech.

```
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;
```



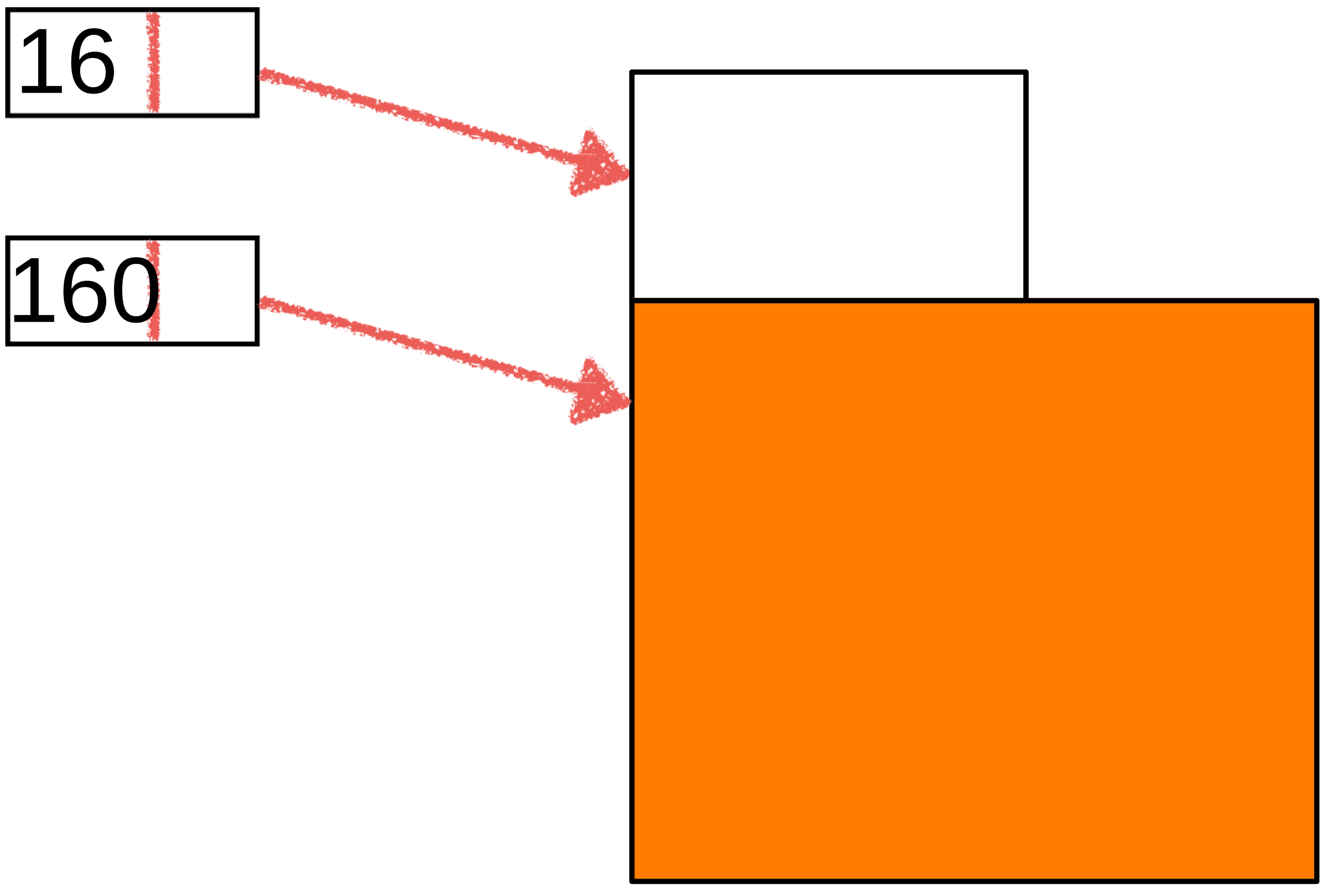
The new-delete mech.

```
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;
```



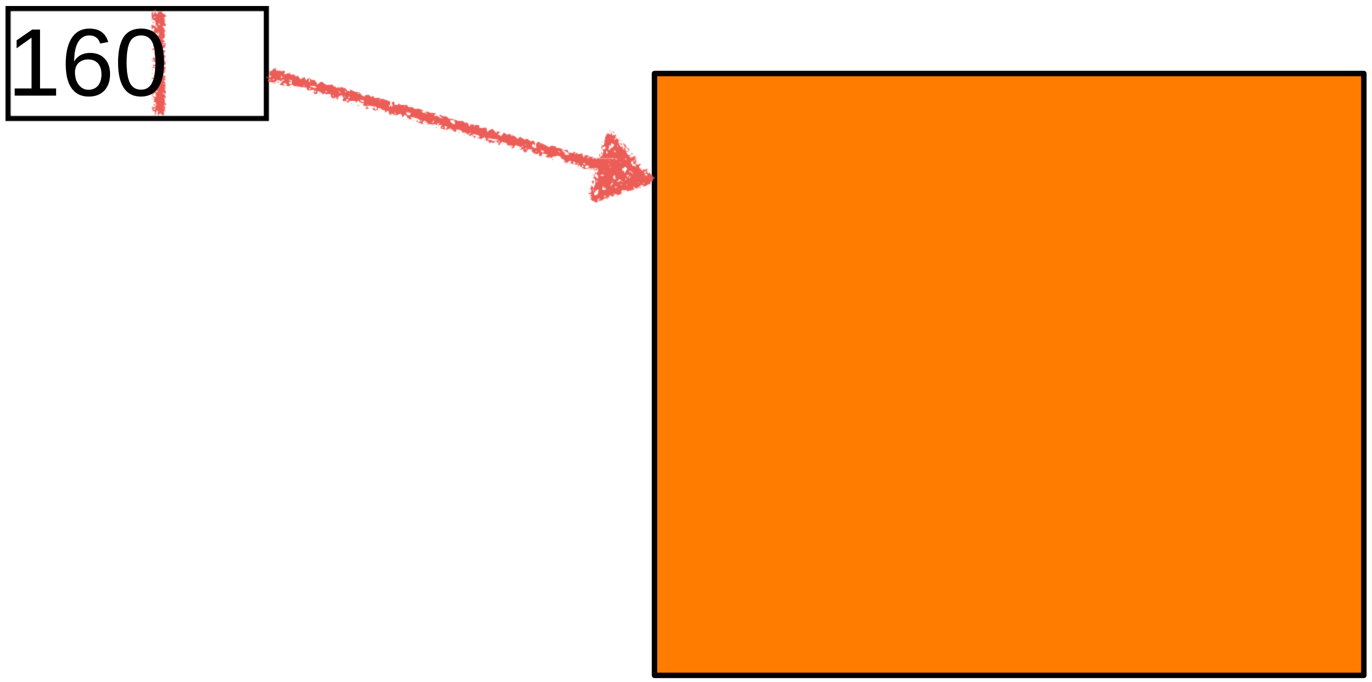
The new-delete mech.

```
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;
```



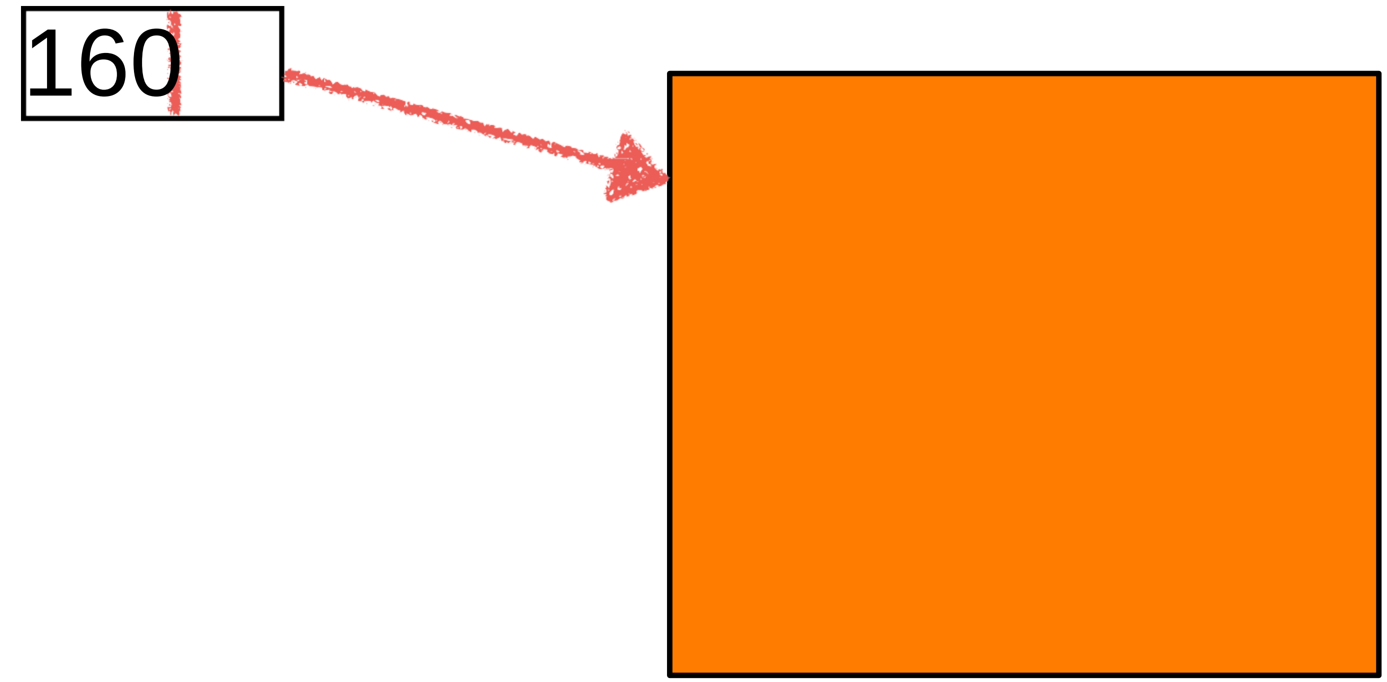
The new-delete mech.

```
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;
```



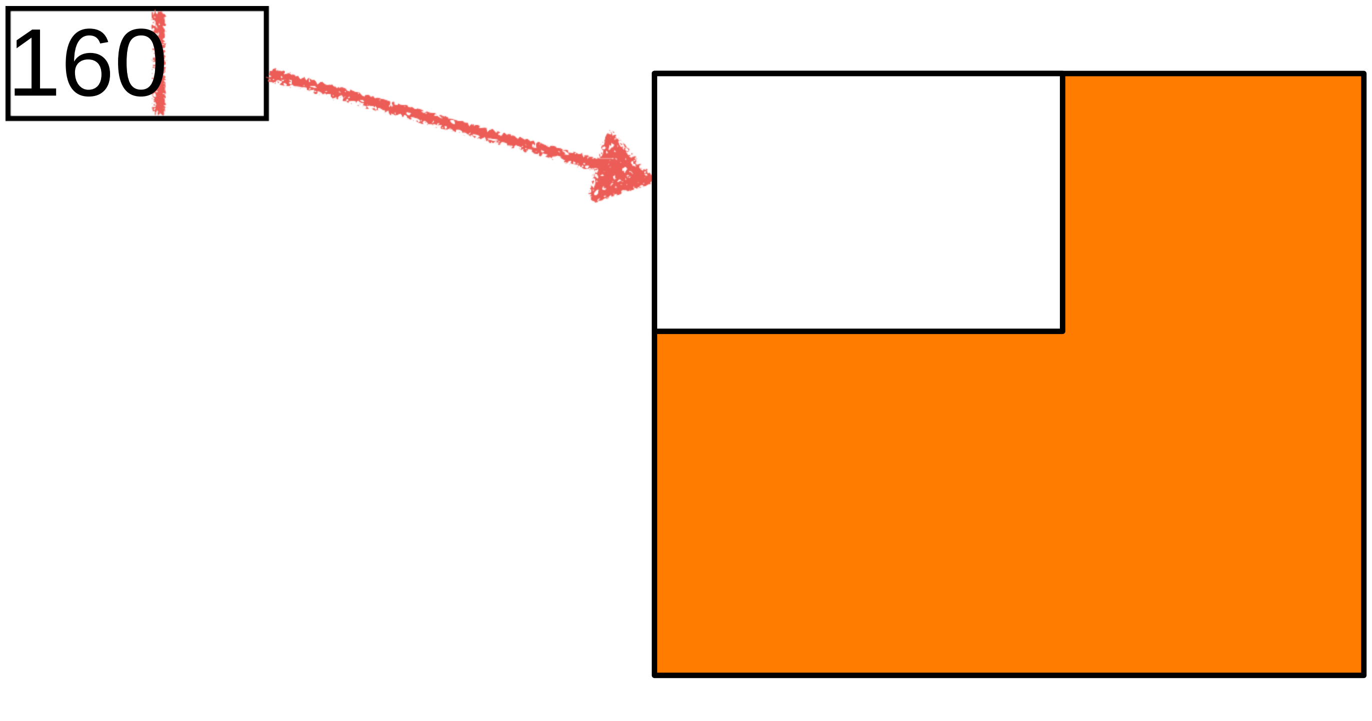
The new-delete mech.

```
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 mech.

```
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;
```

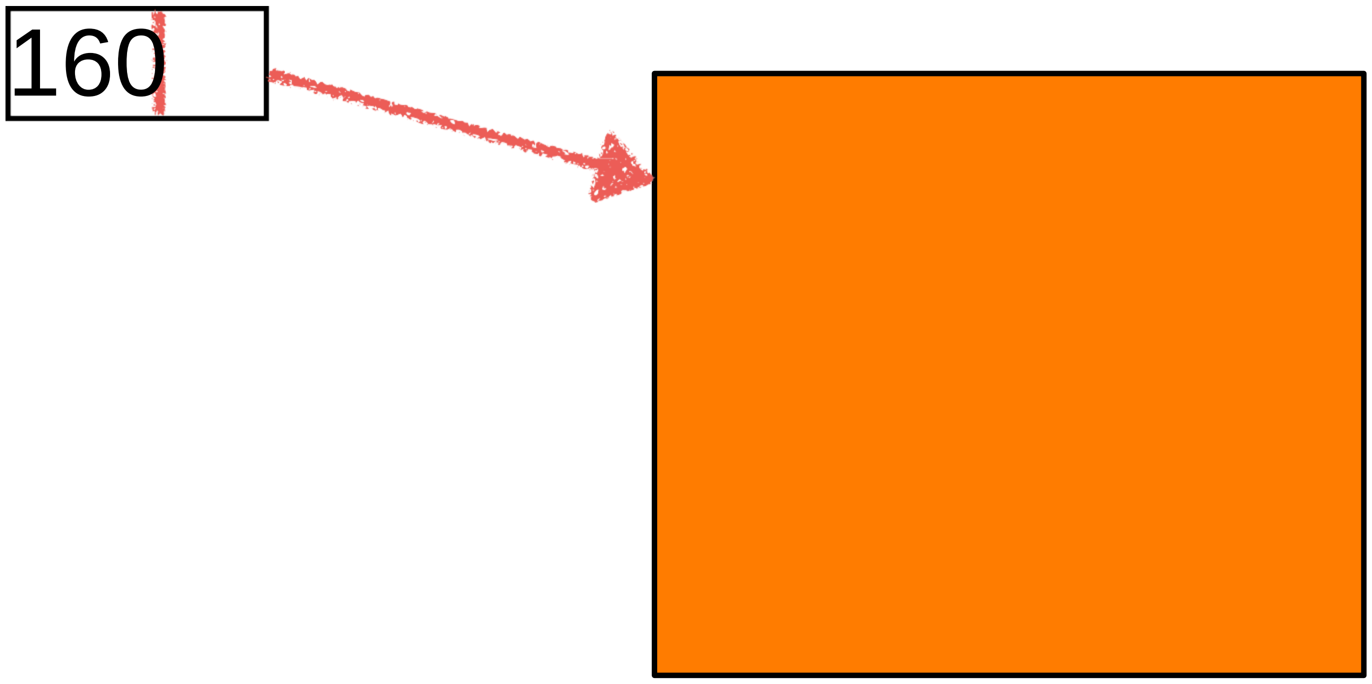


The new-delete mech.

```
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;
```

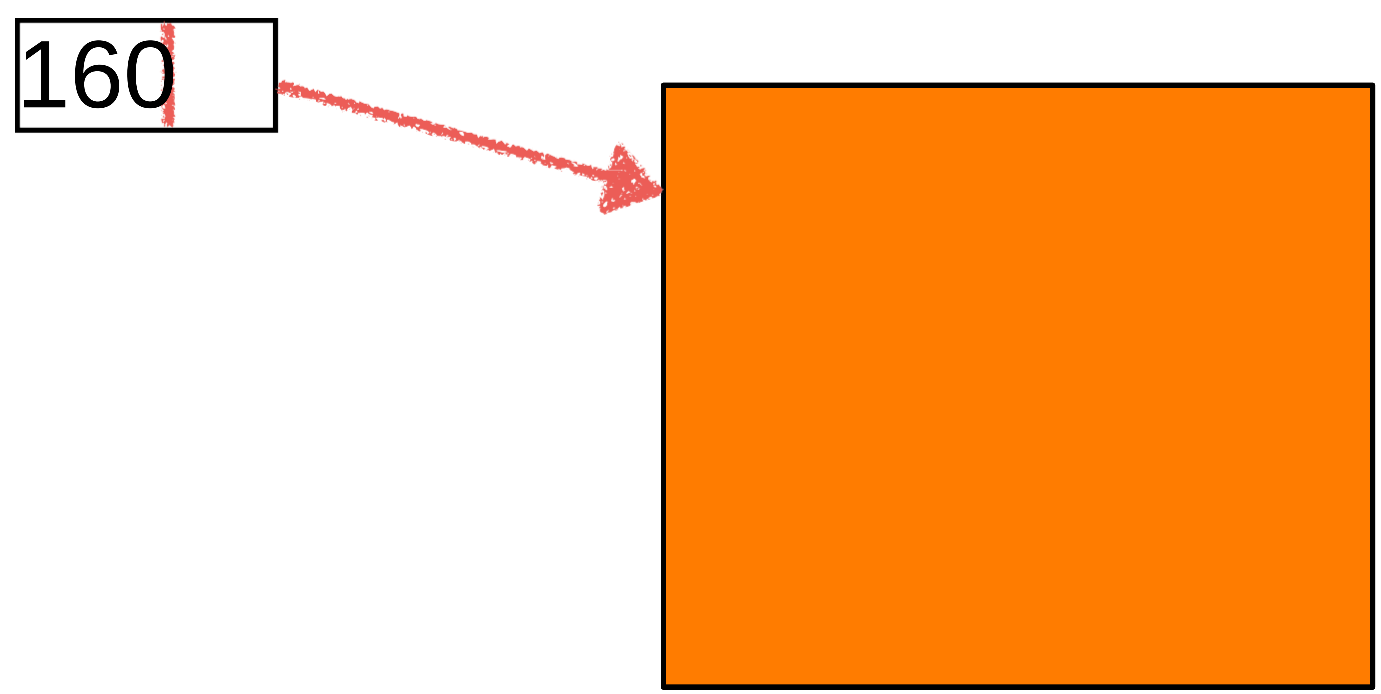
The new-delete mech.

```
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;
```



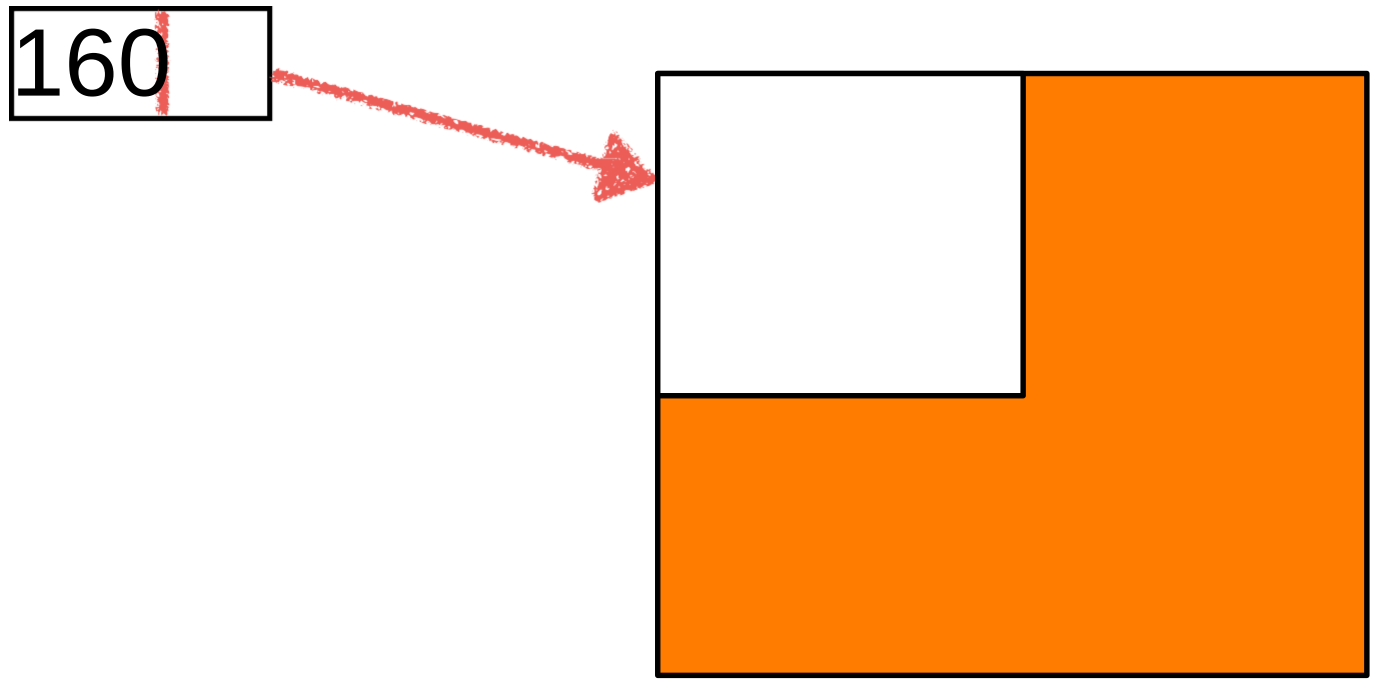
The new-delete mech.

```
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;
```



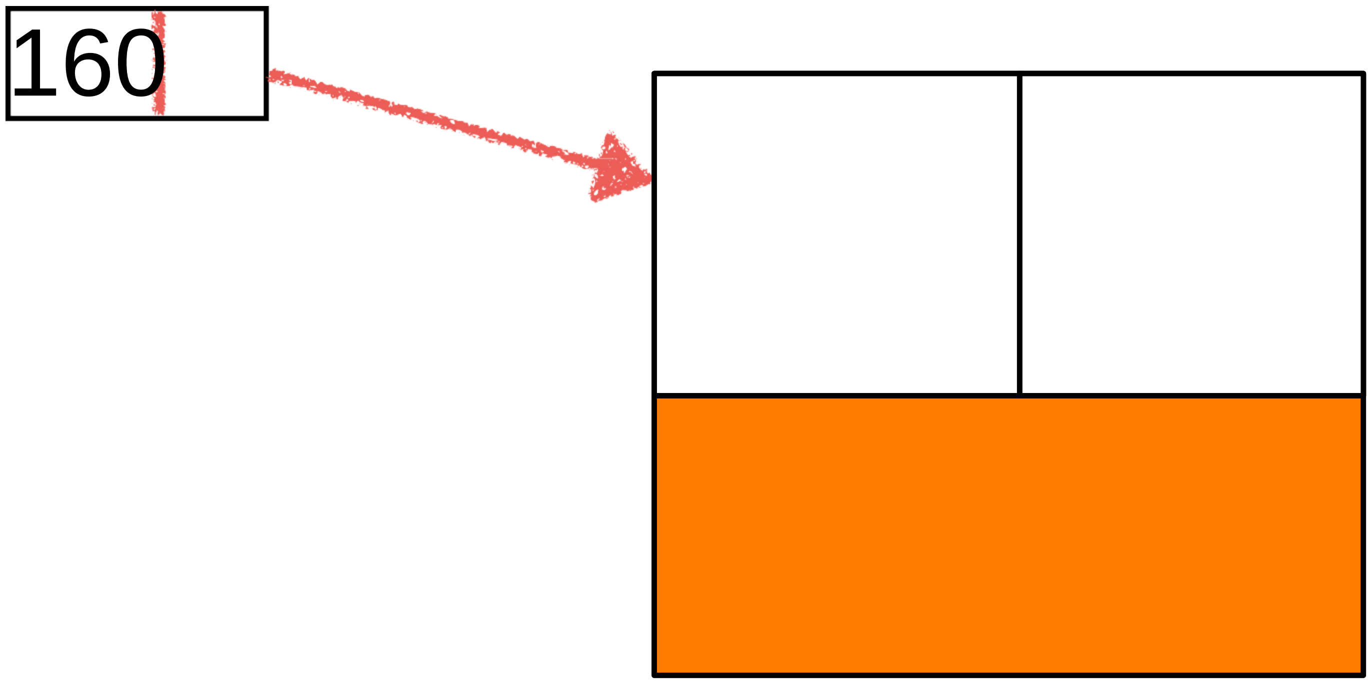
The new-delete mech.

```
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;
```



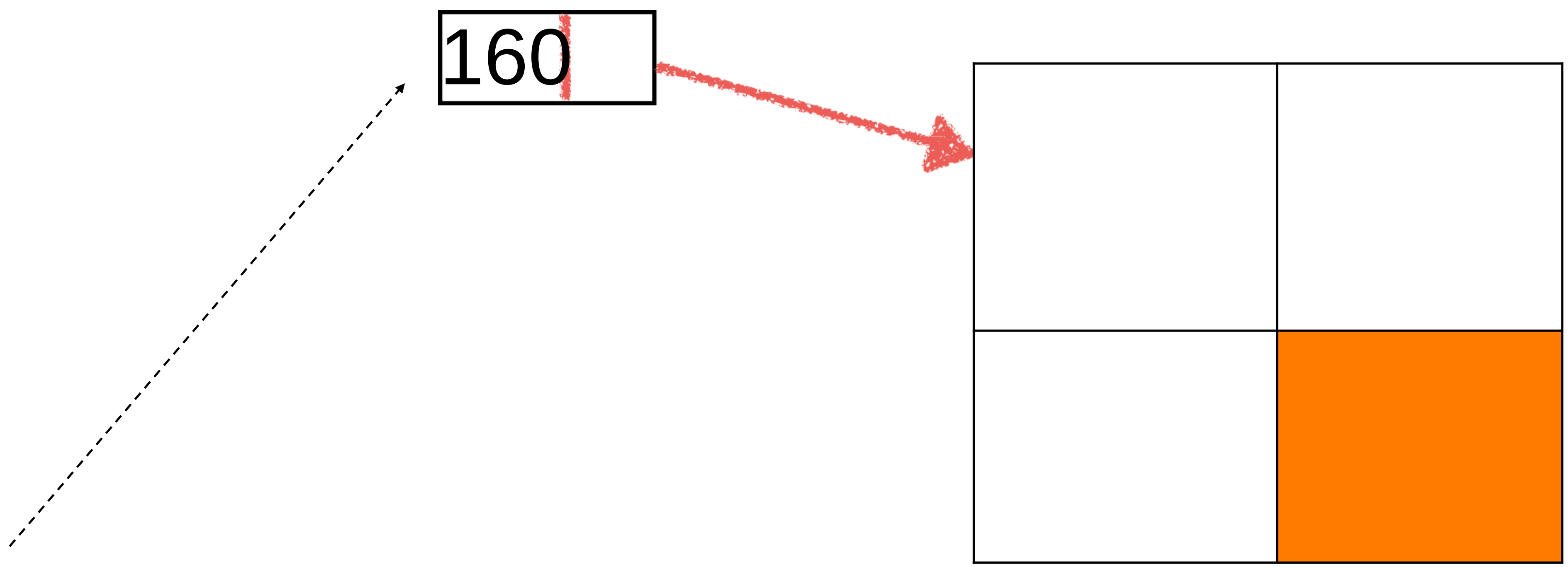
The new-delete mech.

```
int *p=new int;

int *a=new int[10];


Student *q=new Student();


Student *r=new Student[10];


delete p;


a++;delete[] a;


delete q;


delete r;


delete[] r;
```



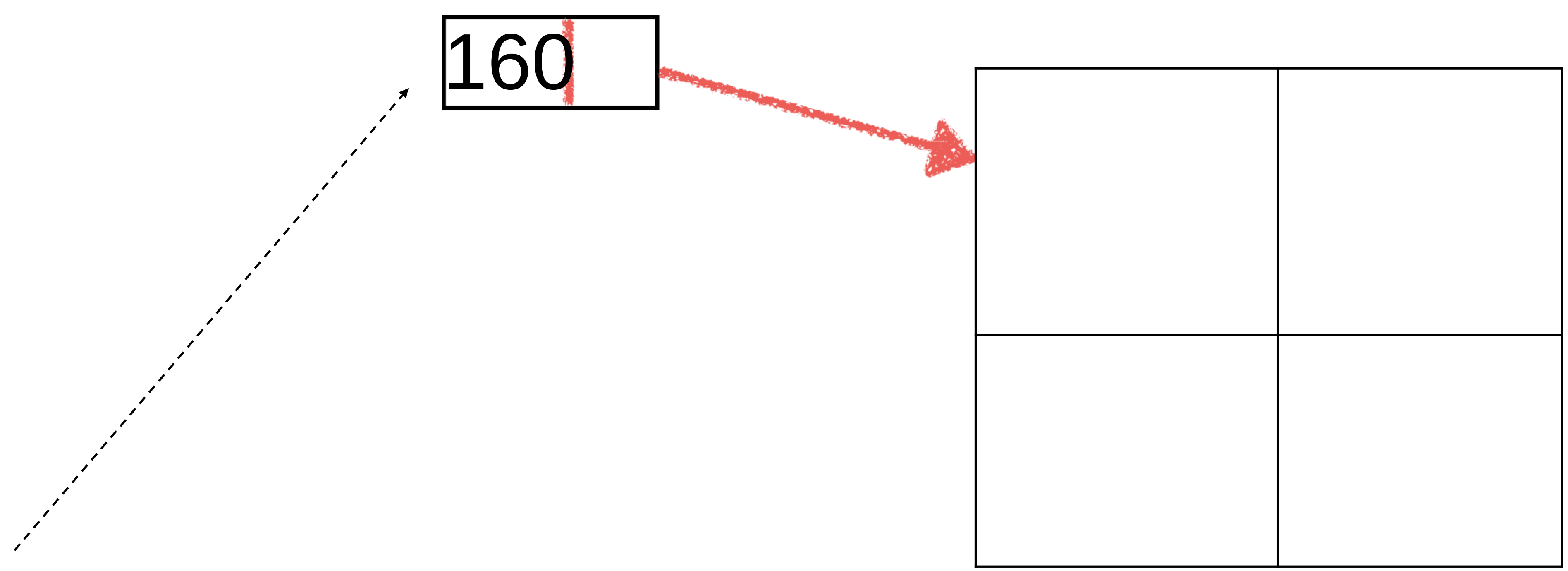
The new-delete mech.

```
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;
```



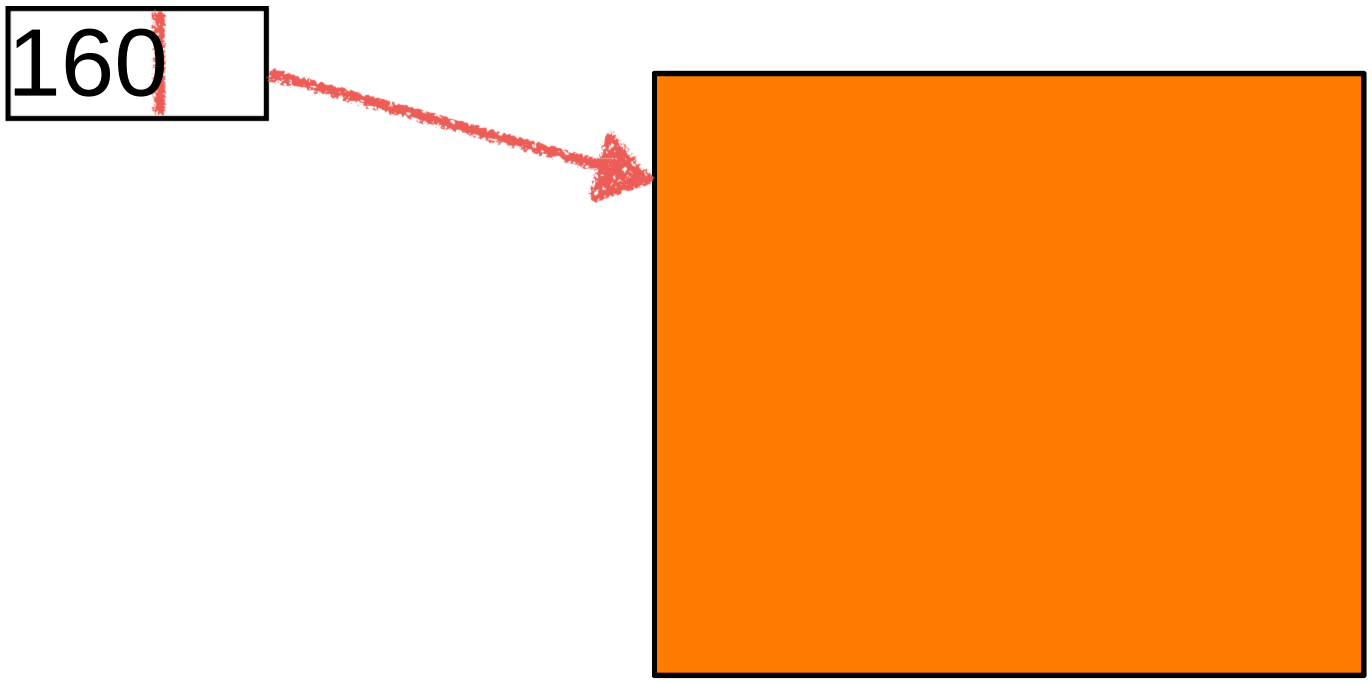
The new-delete mech.

```
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 mech.

```
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).

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

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 declaration

```
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
```

Rules of references

- Bindings don’t change at run time, unlike pointers

Rules of references

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

```
int& y = x;

y = 12; // Changes value of x
```

- The target of a reference must have a location!

```
void func(int &);

func (i * 3);       // Warning or error!
```

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

Restrictions

- No references to references

Restrictions

- No references to references
- No pointers to references

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

  - Reference to pointer is ok

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

- No arrays of references

const

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

Constants

- Constants are 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

```
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

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

Run-time constants

- const value can be exploited

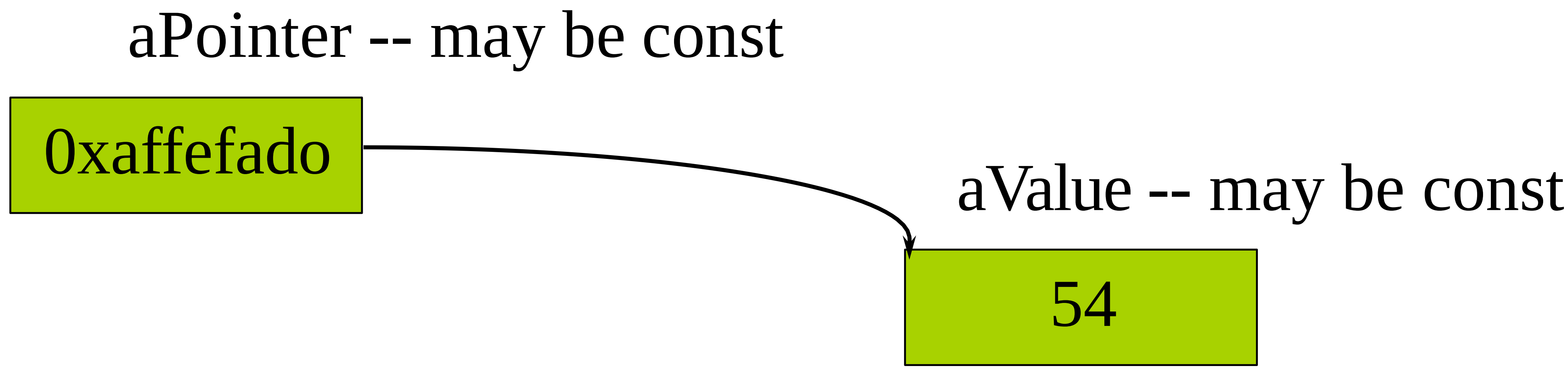
```
const int class_size = 12;
int finalGrade[class_size]; // ok

int x;
cin >> x;
const int size = x;
double classAverage[size]; // error!
```

Aggregates

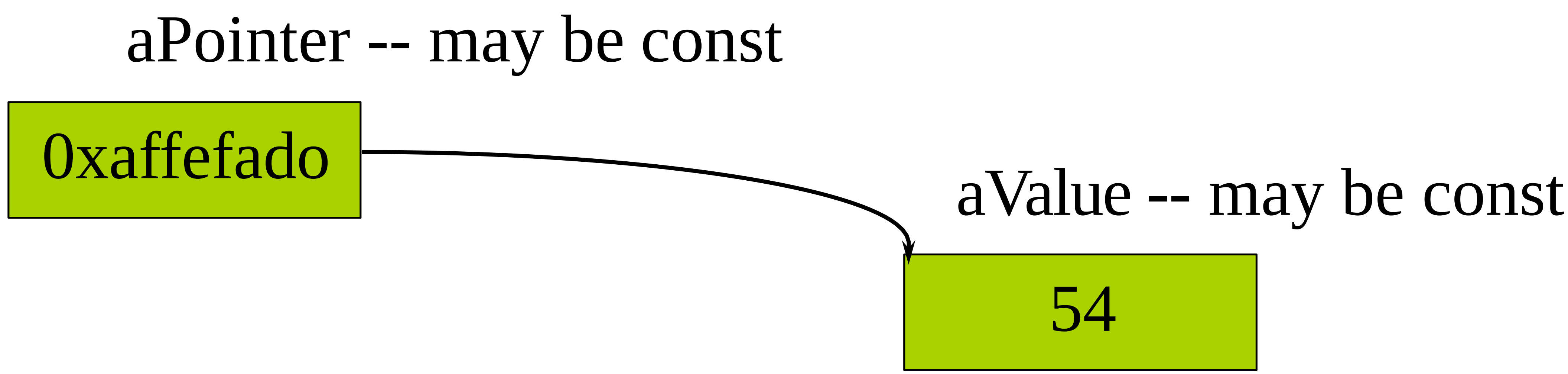
- It's possible to use **const** for aggregates, but storage will be allocated. In these situations, **const** means "a piece of storage that cannot be changed." However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time.

```
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 and const



Pointers and const



- `char * const q = "abc"; //  q is const`
  ```
  *q = 'c'; // OK
  q++;      // ERROR
  ```
- `const char *p = "ABCD";`
  ```
  // (*p) is a const char
  *p = 'b'; // ERROR! (*p) is the const
  ```

Quiz: What do these mean?

```
string p1( “Fred" );
const string* p = &p1;
string const* p = &p1;
string *const p = &p1;
```

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

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?

```
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(); // But this works fine too!

}
```

Passing and returning addresses

- Passing a whole object may cost you a lot. It is better to pass by a pointer. But it’s possible for the programmer to take it and modify the original value.
- In fact, whenever you’re passing an address into a function, you should make it a **const** if at all possible.