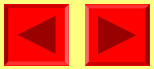
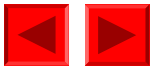
Lesson 5 - Outline

- "`new`" - Dynamic Memory Allocation
- "`delete`" - Dynamic Memory de-Allocation
- Destructors
- Constructors and Destructors: When are they called?
- When to Write an Explicit Destructor



"new" - Dynamic Memory Allocation

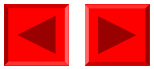
- **new**
  - Creates an object of the proper size
  - Calls its constructor
  - Returns a pointer of the correct type
- Examples of **new**
  - **TypeName *tNPtr;**
    - Creates pointer to a **TypeName** object
  - **tNPtr = new TypeName;**
    - **new** creates **TypeName** object
    - returns pointer which **tNPtr** is set equal to
- Initializing objects
  - **double *dPtr = new double(3.14159);**
    - Initializes object of type **double** to **3.14159**
  - **int *arrayPtr = new int[10];**
    - Creates a ten element **int** array and assigns it to **arrayPtr**



"`delete`" - Dynamic Memory de-Allocation

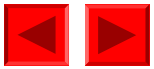
- **`delete`**
  - "*Destroys*" object and frees space

- Examples of **`delete`**
  - **`delete tNPtr;`**
    - "*Destroys*" the **`TypeName`** object and frees its memory area
  - **`delete dPtr;`**
    - "*Destroys*" the **`double`** object and frees its memory area
  - **`delete [] arrayPtr;`**
    - Used to dynamically "*destroy*" array **`arrayPtr`**



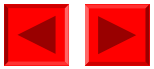
Destructors

- One purpose of a constructor is to provide for the automatic acquisition of a resource.
- Having allocated the resource in the constructor, we need a corresponding operation that **automatically deallocates** or otherwise **releases the resource**.
- The destructor is a *special member function* that can be used to do whatever resource deallocation is needed.
- A destructor serves as the *complement* to the *constructors* of the class.



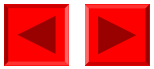
Destructors

- Perform "*termination housekeeping*" before the system reclaims the object's memory
- Name is tilde (**~**) followed by the class name
  - i.e., **`~Time()`**
  - Recall that the constructor's name is the class name
- Receives no parameters, returns no value
- One destructor per class
  - No overloading allowed
- ***Crucial to handle dynamic memory allocation!***



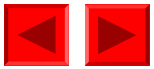
Constructors and Destructors When are they called?

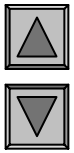
- Constructors and destructors called automatically
  - Order depends on scope of objects
- Global scope objects
  - Constructors called before any other function (including **`main`**)
  - Destructors called when **`main`** terminates (or **`exit`** function called)
  - Destructors not called if program terminates with **`abort`**
- Automatic local objects
  - Constructors called when objects are defined
  - Destructors called when objects leave scope
    - i.e., when the block in which they are defined is exited
  - Destructors not called if the program ends with **`exit`** or **`abort`**



Constructors and Destructors When are they called?

- Static local objects
  - Constructors called when execution reaches the point where the objects are defined
  - Destructors called when **`main`** terminates or the **`exit`** function is called
  - Destructors not called if the program ends with **`abort`**



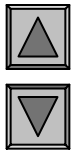


Outline

1. Create a header file

1.1 Include function prototypes for the destructor and constructor

```
1  // Fig. 6.9: create.h
2  // Definition of class CreateAndDestroy.
3  // Member functions defined in create.cpp.
4  #ifndef CREATE_H
5  #define CREATE_H
6
7  class CreateAndDestroy {
8  public:
9     CreateAndDestroy( int );  // constructor
10    ~CreateAndDestroy();      // destructor
11 private:
12    int data;
13 };
14
15 #endif
```



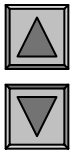
Outline

2. Load the header file

2.1 Modify the constructor and destructor

```
16 // Fig. 6.9: create.cpp
17 // Member function definitions for class CreateAndDestroy
18 #include <iostream>
19
20 using std::cout;
21 using std::endl;
22
23 #include "create.h"
24
25 CreateAndDestroy::CreateAndDestroy( int value )
26 {
27    data = value;
28    cout << "Object " << data << "   constructor";
29 }
30
31 CreateAndDestroy::~CreateAndDestroy()
32    { cout << "Object " << data << "   destructor " << endl; }
```

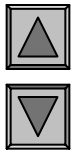
Constructor and Destructor changed to print when they are called.



Outline

3. Create multiple objects of varying types

```
33 // Fig. 6.9: fig06_09.cpp
34 // Demonstrating the order in which constructors and
35 // destructors are called.
36 #include <iostream>
37
38 using std::cout;
39 using std::endl;
40
41 #include "create.h"
42
43 void create( void );   // prototype
44
45 CreateAndDestroy first( 1 );  // global object
46
47 int main()
48 {
49    cout << "   (global created before main)" << endl;
50
51    CreateAndDestroy second( 2 );        // local object
52    cout << "   (local automatic in main)" << endl;
53
54    static CreateAndDestroy third( 3 );  // local object
55    cout << "   (local static in main)" << endl;
56
57    create();  // call function to create objects
58
59    CreateAndDestroy fourth( 4 );        // local object
60    cout << "   (local automatic in main)" << endl;
61    return 0;
62 }
```



Outline

```
63
64 // Function to create objects
65 void create( void )
66 {
67    CreateAndDestroy fifth( 5 );
68    cout << "   (local automatic in create)" << endl;
69
70    static CreateAndDestroy sixth( 6 );
71    cout << "   (local static in create)" << endl;
72
73    CreateAndDestroy seventh( 7 );
74    cout << "   (local automatic in create)" << endl;
75 }
```

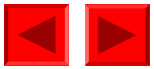
Program Output

```
OUTPUT
Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor
```

Notice how the order of the constructor and destructor call depends on the types of variables (automatic, global and **`static`**) they are associated with.

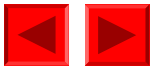
When to Write an Explicit Destructor

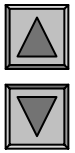
- Many classes do not require an explicit destructor.
  - In particular, a class that has a constructor does not necessarily need to define its own destructor.
- Destructors are needed only if there is work for them to do.
  - A destructor is not limited only to relinquishing resources. A destructor, in general, can perform any operation that the class designer wishes to have executed subsequent to the last use of an object of that class.



When to Write an Explicit Destructor

- Ordinarily they are used to relinquish resources acquired in the constructor or during the lifetime of the object.
- A useful *rule of thumb* is that if a class needs a destructor, it will also need the *assignment operator* and a *copy constructor*.
  - This rule is often referred to as the *Rule of Three*, indicating that if you need a destructor, then you need all three copy-control members.

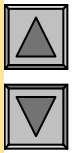




Outline


```
1  // Fig. 7.9: employ1.h
2  // An employee class
3  #ifndef EMPLOY1_H
4  #define EMPLOY1_H
5
6  class Employee {
7  public:
8     Employee( const char*, const char* );  // constructor
9     ~Employee();                           // destructor
10    const char *getFirstName() const;  // return first name
11    const char *getLastName() const;   // return last name
12
13    // static member function
14    static int getCount();  // return # objects instantiated
15
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data member
21    static int count;  // number of objects instantiated
22 };
23
24 #endif
```

`static` member function and variable declared.



Outline

1. Load header file

1.1 Initialize `static` data members

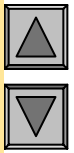
1.2 Function definitions

```
25 // Fig. 7.9: employ1.cpp
26 // Member function definitions for class Employee
27 #include <iostream>
28
29 using std::cout;
30 using std::endl;
31
32 #include <cstring>
33 #include <cassert>
34 #include "employ1.h"
35
36 // Initialize the static data member
37 int Employee::count = 0;
38
39 // Define the static member function that
40 // returns the number of employee objects instantiated.
41 int Employee::getCount() { return count; }
42
43 // Constructor dynamically allocates space for
44 // first and last name and uses strcpy to copy
45 // the first and last names into the object
46 Employee::Employee( const char *first, const char *last )
47 {
48    firstName = new char[ strlen( first ) + 1 ];
49    assert( firstName != 0 );   // ensure memory allocated
50    strcpy( firstName, first );
51
52    lastName = new char[ strlen( last ) + 1 ];
53    assert( lastName != 0 );    // ensure memory allocated
54    strcpy( lastName, last );
55
56    ++count;  // increment static count of employees
```

`static` data member **`count`** and function **`getCount( )`** initialized at file scope (required).

Note the use of **`assert`** to test for memory allocation.

`static` data member **`count`** changed when a constructor/destructor called.

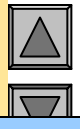


Outline


```
57    cout << "Employee constructor for " << firstName
58         << ' ' << lastName << " called." << endl;
59 }
60
61 // Destructor deallocates dynamically allocated memory
62 Employee::~Employee()
63 {
64    cout << "~Employee() called for " << firstName
65         << ' ' << lastName << endl;
66    delete [] firstName;  // recapture memory
67    delete [] lastName;   // recapture memory
68    --count;  // decrement static count of employees
69 }
70
71 // Return first name of employee
72 const char *Employee::getFirstName() const
73 {
74    // Const before return type prevents client from modifying
75    // private data. Client should copy returned string before
76    // destructor deletes storage to prevent undefined pointer.
77    return firstName;
78 }
79
80 // Return last name of employee
81 const char *Employee::getLastName() const
82 {
83    // Const before return type prevents client
84    // private data. Client should copy returned
85    // destructor deletes storage to prevent und
86    return lastName;
87 }
```

`static` data member **`count`** changed when a constructor/destructor called.

`Count` decremented because of destructor calls from **`delete`**.



Outline

```
88 // Fig. 7.9: fig07_09.cpp
89 // Driver to test the employee class
90 #include <iostream>
91
92 usi
93 usi
94
95 #inc
96
97 int main()
98 {
99    cout << "Number of employees before ins
100        << Employee::getCount() << endl;
101
102   Employee *e1Ptr = new Employee( "Susan", "Baker" );
103   Employee *e2Ptr = new Employee( "Robert
104
105   cout << "Number of employees after instantiation is "
106        << e1Ptr->getCount();
107
108   cout << "\n\nEmployee 1: "
109        << e1Ptr->getFirstName()
110        << " " << e1Ptr->getLastName()
111        << "\nEmployee 2: "
112        << e2Ptr->getFirstName()
113        << " " << e2Ptr->getLastName() << "\n\n";
114
115   delete e1Ptr;   // recapture memory
116   e1Ptr = 0;
117   delete e2Ptr;   // recapture memory
118   e2Ptr = 0;
```

count incremented because of constructor calls from **new**.

If no **Employee** objects exist **getCount** must be accessed using the class name and (**::**).

```
Number of employees before instantiation is 0
```

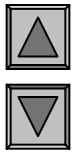
e2Ptr->getCount() or **Employee::getCount()** would also work.

```
Number of employees after instantiation is 2
```

```
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
```

```
Employee 1: Susan Baker
Employee 2: Robert Jones
```

```
~Employee() called for Susan Baker
~Employee() called for Robert Jones
```



Outline

```
119
120   cout << "Number of employees after deletion is "
121        << Employee::getCount() << endl;
122
123   return 0;
124}
```

count back to zero.

Program Output

```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```