600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

*Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.
Outline

• Review
  • Constructors, destructors

• new and delete
Default Constructors

• The default constructor is called when no initialization parameters are passed

```
main.cpp
#include "instructor.h"
int main( void )
{
    Instructor inst; // Default constructor called
    ...
}
```

```
instructor.h
#ifndef INSTRUCTOR_INCLUDED
#define INSTRUCTOR_INCLUDED
#include <string>
class Instructor
{
public:
    std::string name;
    void print( void ) const;
};
#endif // INSTRUCTOR_INCLUDED
```
Default Constructors

• The *default constructor* is called when no initialization parameters are passed
  • If no constructor is given, C++ implicitly defines one which calls the default constructor of each of the members

```
#ifndef INSTRUCTOR_INCLUDED
#define INSTRUCTOR_INCLUDED
#include <string>
class Instructor
{
public:
    std::string name;
    void print( void ) const;
};
#endif // INSTRUCTOR_INCLUDED
```
Default Constructors

• The *default constructor* is called when no initialization parameters are passed
  • Or the class can provide its own
    • Looks like a function:
      • Whose name is the class name
      • With no (void) arguments
      • With no return type
    • This should be public

```
instructor.h
 ifndef INSTRUCTOR_INCLUDED
#define INSTRUCTOR_INCLUDED
#include <string>
class Instructor
{
 public:
   std::string name;
   Instructor( void ){ name = "misha"; }
   void print( void ) const;
};
#endif // INSTRUCTOR_INCLUDED
```
Default Constructors

• The default constructor is called when no initialization parameters are passed
  • Or the class can provide its own
    • It can be defined in the class definition (if it's short)

instructor.h

#ifdef INSTRUCTOR_INCLUDED
#define INSTRUCTOR_INCLUDED
#include <string>
class Instructor
{
    public:
        std::string name;
        Instructor( void ){ name = "misha"; }
        void print( void ) const;
};
#endif // INSTRUCTOR_INCLUDED
Default Constructors

• The default constructor is called when no initialization parameters are passed
  • Or the class can provide its own
    • It can be defined in the class definition (if it's short)
    • Or it can be declared in the .h file and defined in the .cpp file

```
#include "instructor.h"
Instructor::Instructor( void ){ name = "misha"; }
```

```
instructor.h
#ifndef INSTRUCTOR_INCLUDED
#define INSTRUCTOR_INCLUDED
#include <string>
class Instructor {
 public:
   std::string name;
   Instructor( void )
      void print( void ) const;
};
#endif // INSTRUCTOR_INCLUDED
```
Non-Default Constructors

• Constructors can also take arguments, allowing the caller to "customize" the object

```cpp
#include <iostream>
#include "instructor.h"

int main( void )
{
    Instructor i1, i2("yotam");
    std::cout << i1.name << std::endl;
    std::cout << i2.name << std::endl;
    return 0;
}
```

```cpp
#ifndef INSTRUCTOR_INCLUDED
#define INSTRUCTOR_INCLUDED
#include <string>

class Instructor
{
public:
    std::string name;
    Instructor( void ){ name = "misha"; }
    Instructor( std::string n ){ name = n; }
    void print( void ) const;
};
#endif // INSTRUCTOR_INCLUDED
```
Non-Default Constructors

• Before the object's constructor is called, C++ calls the default constructor of each member

```cpp
#include "myClass.h"
int main( void )
{
    C2 c2( 5 );
    return 0;
}
```

```cpp
myClass.h
#include <iostream>
class C1{
    int _value;
    void _init( int v )
    {
        _value = v;
        std::cout << "C1( " << v << " )" << std::endl;
        // Do a lot of work here
    }
public:
    C1( void ){_init(0); }
    C1( int v ){_init(v); }
};
class C2{
    C1 _c1;
public:
    C2( int v ){
        std::cout << "C2( " << v << " )" << std::endl;
        _c1 = C1( v );
    }
};
```
Non-Default Constructors

• Before the object's constructor is called, C++ calls the default constructor of each member
  • We call C1's default constructor and then its non-default constructor
  • We would like to invoke C1's non-default constructor directly

```cpp
class C1
{
    int _value;
    void _init( int v )
    {
        _value = v;
        std::cout << "C1( " << v << " )" << std::endl;
        // Do a lot of work here
    }
public:
    C1( void ){ _init(0); }  // Default constructor
    C1( int v ){ _init(v); } // Parameterized constructor
};

# include <iostream>
#include "myClass.h"
class C2
{
    C1 _c1;
public:
    C2( int v )
    {
        std::cout << "C2( " << v << " )" << std::endl;
        _c1 = C1( v );
    }
};

class C1
{
    int _value;
    void _init( int v )
    {
        _value = v;
        std::cout << "C1( " << v << " )" << std::endl;
        // Do a lot of work here
    }
public:
    C1( void ){ _init(0); }  // Default constructor
    C1( int v ){ _init(v); } // Parameterized constructor
};

#include <iostream>
#include "myClass.h"
class C2
{
    C1 _c1;
public:
    C2( int v )
    {
        std::cout << "C2( " << v << " )" << std::endl;
        _c1 = C1( v );
    }
};
```

```cpp
#include "myClass.h"
int main( void )
{
    C2 c2( 5 );
    return 0;
}
```
Non-Default Constructors

*Initializer lists* allow us to specify that a (non-default) constructor should be used to initialize the member directly

- Before the body of the constructor:
  - a "::" followed by a comma-separated list of member constructors

```cpp
#include "myClass.h"
int main( void )
{
    C2 c2( 5 );
    return 0;
}
```

```cpp
myClass.h
#include <iostream>
class C1
{
    int _value;
    void _init( int v )
    {
        _value = v;
        std::cout << "C1( " << v << ")" << std::endl;
        // Do a lot of work here
    }
public:
    C1( void ){ _init(0); }
    C1( int v ){ _init(v); }
};
class C2
{
    C1 _c1;
public:
    C2( int v ): _c1(v)
    {
        std::cout << "C2( " << v << ")" << std::endl;
    }
};
```
Non-Default Constructors

• *Initializer lists* allow us to specify that a (non-default) constructor should be used to initialize the member directly
  
  • They can be used to initialize primitive types (even though those don’t have a constructor)

```cpp
#include "myClass.h"
int main( void )
{
    C2 c2( 5 );
    C1(5)
    C2(5)
    return 0;
}
```
Destructors

• A class *constructor*’s job is to initialize the fields of the object
  • It’s common for a constructor to obtain a resource (allocate memory, open a file, etc.) that should be released when the object is destroyed

• A class *destructor* is a method called by C++ when the object goes out of scope or is otherwise deallocated
Destructors

• A class constructor’s job is to initialize the object:
  • It’s common for a constructor to obtain a resource (allocate memory, open a file, etc.) that should be released when the object is destroyed.

• A class destructor is a method called by C++ when the object goes out of scope or is otherwise deallocated (e.g., using delete):
  • Looks like a function:
    • Whose name is the class name
    • prepended with a "~"
  • With no (void) arguments
  • With no return type
  • This should be public

```cpp
defmain.cpp
#include <iostream>
#include <cassert>
class Foo {
public:
    size_t sz;
    int* values;
    Foo( int s ) : sz( s ) {
        values = malloc( sizeof( int )*sz );
        assert( values );
    }
    ~Foo(void) { free( values ); }
};
int main( void ) {
    Foo f( 10 );
    return 0;
}
```
Destructors

• A class constructor’s job is to initialize the object.
  • It’s common for a constructor to obtain a resource (allocate memory, open a file, etc.) that should be released when the object is destroyed.

• A class destructor is a method called by C++ when the object goes out of scope or is otherwise deallocated (e.g., using `delete`).
  • As with other methods, it can be defined in the class definition or outside of it.

```cpp
#include <iostream>
#include <cassert>

class Foo
{
  public:
    size_t sz;
    int* values;
    Foo( int s ) : sz( s )
    {
      values = malloc( sizeof(int)*sz );
      assert( values );
    }
    ~Foo( void );
};

Foo::~Foo( void ){ free( values ); }

int main( void )
{
  Foo f( 10 );
  return 0;
}
```
Outline

• Review
• new and delete
Memory Allocation

• In C, we allocate memory on the heap using `malloc`:
  ```c
  void* malloc( size_t size );
  ```
  • This function does not need to know the type of the data
  • Just the size of the memory we were requesting

• Similarly, we deallocate memory from the heap using `free`:
  ```c
  void free( void* ptr );
  ```
  • This function does not need to know the type of the data
  • Just the location that we are freeing
Memory Allocation

• In C++, we need to know the data-type to invoke the constructor
• We do this using the `new` operator:

  `<DataType>* new DataType( <ConstructorParams> );`

  • This allocates memory for a single object
  • And it invokes the constructor
    • If no constructor parameters are passed, the default constructor is invoked
    • Though primitive types (e.g. ints, chars, etc.) don't have constructors, we can use `new` to allocate them on the heap

```cpp
main.cpp
#include <iostream>
#include <string>
using std::string;
int main( void )
{
    string* strPtr = new string( "Hello" );
    std::cout << *strPtr << std::endl;
    ...
    return 0;
}
```
Memory Allocation

• In C++, we need to know the data-type to invoke the destructor
• We do this using the `delete` operator:

```
delete <DataType>*;
```

• This invokes the destructor of the object
  • Though primitive types (e.g. ints, chars, etc.) don't have destructors, we can use `delete` to deallocate them from the heap
• And deallocates its memory

```cpp
#include <iostream>
#include <string>
using std::string;
int main( void )
{
    string* strPtr = new string( "Hello" );
    std::cout << *strPtr << std::endl;
    delete strPtr;
    return 0;
}
```
Memory Allocation

• We allocate multiple objects simultaneously using `new[]`:
  ```cpp
  <DataType>* new DataType[<NumElems>];
  ```
  • This allocates memory for a `NumElems` objects
  • And it invokes the default constructor for each one

• And we deallocate using `delete[]`:
  ```cpp
  delete[] <DataType>*;
  ```
  • This invokes the destructor for each object
  • And then deallocates the memory for the entire array of objects
Memory Allocation

```cpp
#include <iostream>
#include "instructor.h"
using namespace std;
int main( void )
{
    int count;
    cout << "Instructor Count: ";
    cin >> count;
    Instructor** instr = new Instructor*[count];
    for( int i=0 ; i<count ; i++ )
    {
        cout << "Name: ";
        string name;
        cin >> name;
        instr[i] = new Instructor( name );
    }
    for( int i=0 ; i<count ; i++ ) cout << instr[i]->name << endl;
    for( int i=0 ; i<count ; i++ ) delete instr[i];
    delete[] instr;
    return 0;
};
```
Memory Allocation

- Memory allocated with `new` must be deallocated with `delete`
Memory Allocation

- Memory allocated with `new` must be deallocated with `delete`

- Memory allocated with `new[]` must be deallocated with `delete[]`
  - We are deleting an array of pointers to `Instructor` objects
  ⇒ The `Instructor` destructor is not called

```cpp
#include <iostream>
#include "instructor.h"
using namespace std;

int main( void )
{
    int count;
    cout << "Instructor Count: ";
    cin >> count;
    Instructor** instr = new Instructor*[count];
    for( int i=0 ; i<count ; i++ )
    {
        cout << "Name: ";
        string name;
        cin >> name;
        instr[i] = new Instructor( name );
    }
    for( int i=0 ; i<count ; i++ ) cout << instr[i]->name << endl;
    for( int i=0 ; i<count ; i++ ) delete instr[i];
    delete[] instr;
    return 0;
}
```
Piazza → Resources section → Resources tab → Exercise 9-1