ACCELERATED COMPUTER SCIENCE FUNDAMENTALS

Matt Richards

March 1, 2021

TITLE: Accelerated Computer Science Fundamentals
AUTHOR: Matt Richards
NUMBER OF PAGES: 73

Contents

Chapter 1

Object Oriented Programming Fundamentals with C++

1.1 Introduction to Classes in C++

C++ is a strongly typed programming language meaning each variable has a type, name, value, and location in memory. There are only two types of variables, primitives and User-defined. Some primitives are: int, char, bool, float, double, and void. There are however an unlimited number of User-defined type. Two of the most common user-defined types are std::string, and std::vector. I will not go through fundamental syntax and C++ programming because I understand that for myself.

Definition: C++ classes - The C++ class encapsulates data and it associates functionality into an object.

Definition: Encapsulation - Encapsulation encloses data and functionality into a single unit (called a class). This is illustrated in Figure 1.1.

- In C++ and Python, data and functionality are separated into two separate protections called public and private.
- Client code: any code that import our class and makes use of the data type in the class.
- Public members can be accessed by client code.
- Private members can not be accessed by client code (only used within the class itself).

We can organize our code easier using header files and Makefiles which I am already familiar with. Much of the code in this course is written using Makefiles to make compiling the program much simpler.

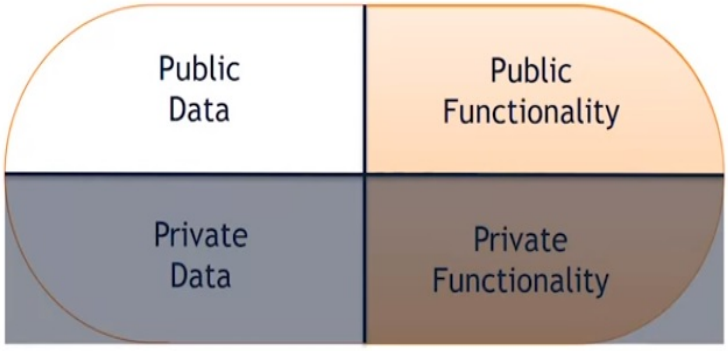

Figure 1.1: Illustration of the concept of encapsulation (the class is the pill and all the content is encapsulated inside).

Listing 1.1: Writing the header file (.h), the implementation file (.cpp), and the main file

```
/**
 * Simple C++ class for representing a Cube.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

// All header (.h) files start with "#pragma once":
#pragma once

// A class is defined with the 'class' keyword, the name
// of the class, curly braces, and a required semicolon
// at the end:
class Cube {
  public:  // Public members:
    double getVolume();
    double getSurfaceArea();
    void setLength(double length);

  private: // Private members:
    double length_;
};
```

```
/**
 * Simple C++ class for representing a Cube.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include "Cube.h"

double Cube::getVolume() {
  return length_ * length_ * length_;
}

double Cube::getSurfaceArea() {
  return 6 * length_ * length_;
}

void Cube::setLength(double length) {
  length_ = length;
}

/**
 * C++ code for creating a Cube of length 2.4 units.
 * - See ../cpp-std/main.cpp for a similar program with print statements.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include <iostream>
#include "Cube.h"

int main() {
  Cube c;

  c.setLength(3.48);
  double volume = c.getVolume();
  std::cout << "Volume: " << volume << std::endl;
```

```
  return 0;
}
```

1.1.1 C++ Standard Library (std)

The C++ standard library (std) provides a set of commonly used functionality and data tools to build upon. #include < iostream > (in/out stream) allows one to write programs which write/read to the console, and write/read to separate files. I will not go over the libraries that I am already familiar with.

All functionality used for the standard library will be part of the std namespace. These namespaces allow us to avoid name conflict for commonly used names. The command "using namespace::std" does this for us.

We can actually create our own namespace's through a class structure. The reason we might do this is because calling a class named "cube" is rather ambiguous and could lead to conflicts with larger programs. This is shown in Listing 1.2.

Listing 1.2: Implementing the std library of C++.

```
//We can create our own namespace's through a class structure,
//In the form represented below.
/**
 * Simple C++ class for representing a Cube.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#pragma once

namespace uiuc {
  class Cube {
    public:
      double getVolume();
      double getSurfaceArea();
      void setLength(double length);

    private:
      double length_;
```

```
  };
}

/**
 * Simple C++ class for representing a Cube.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include "Cube.h"

namespace uiuc {
  double Cube::getVolume() {
    return length_ * length_ * length_;
  }

  double Cube::getSurfaceArea() {
    return 6 * length_ * length_;
  }

  void Cube::setLength(double length) {
    length_ = length;
  }
}

/**
 * Simple C++ making use of std::cout and a 'Cube' class.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include <iostream>
#include "Cube.h"

int main() {
  uiuc::Cube c;
```

```
  c.setLength(2.4);
  std::cout << "Volume: " << c.getVolume() << std::endl;

  double surfaceArea = c.getSurfaceArea();
  std::cout << "Surface Area: " << surfaceArea << std::endl;

  return 0;
}
```

1.2 Memory in C++

In C++, the programmer has control over the memory and life cycle of every variable. By default, all variables live in stack memory. Every variable has a name, type, value, and location in memory. In C++, the & operator returns the memory address of a variable.

1.2.1 Stack Memory

By default, every variable in C++ is placed in Stack memory.

Definition: Stack memory - Stack memory is associated with the current function and the memory's life cycle is tied to the function:

- When the function returns or ends, the stack memory of that function is released.
- Stack memory always starts from high addresses and grows down.

Listing 1.3: Stack memory examples.

```
/**
 * C++ program printing memory addresses of variables across two functions.
 *
 * @author
 *    Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include <iostream>

void foo() {
  int x = 42;
```

```
  std::cout << " x in foo(): " <<  x << std::endl;
  std::cout << "&x in foo(): " << &x << std::endl;
}

int main() {
  int num = 7;
  std::cout << " num in main(): "  << num << std::endl;
  std::cout << "&num in main(): " << &num << std::endl;
  //num will have a high address value.

  foo();
  //foo will have allocation before num and it "stacks" below it.
  //Generally speaking, stack memory goes from highest allocation to zero.

  //address of num is 0x7ffcaf2c393c
  //address of foo is 0x7ffcaf2c391c

  return 0;
}
```

1.2.2 Pointers

The way we can utilize a memories address is with pointers.

Definition: Pointer - A pointer is a variable that stores the memory address of the data. We define a pointer by adding a * to the type of the variable.

- We can think of pointers as a level of indirection from the data.
- Its syntax would look like (int * p = & num;)

Dereference Operator: The dereference operator is denoted as "*" in C++. It operates on a pointer and returns the value at the pointers address.

Given a pointer, a level of indirection can be removed by proceeding the variable with an asterisk. This is shown in Listing 1.4. We can visualize how this stack memory is being allocated with pointers through Figure 1.2. Here we draw from the cube class we created before in Listing 1.2.

Listing 1.4: Pointer stuff in C++

```
//Dereferencing an operator
```

```
int num = 7;
//We declare a variable "num" to have a value of 7.
// "num" has its own address, we don't know what that is.
int * p = &num;
// p is declared a pointer and it points to the address of num.
int value_in_num = *p;
//value_in_num is a third variable and it takes on the dereferenced value of p.
//Once we see what p is pointing to, we go ahead
//and copy that value into value_in_num.
*p = 42;
//The last line declares the dereferenced value in p, to a value of 42.
```

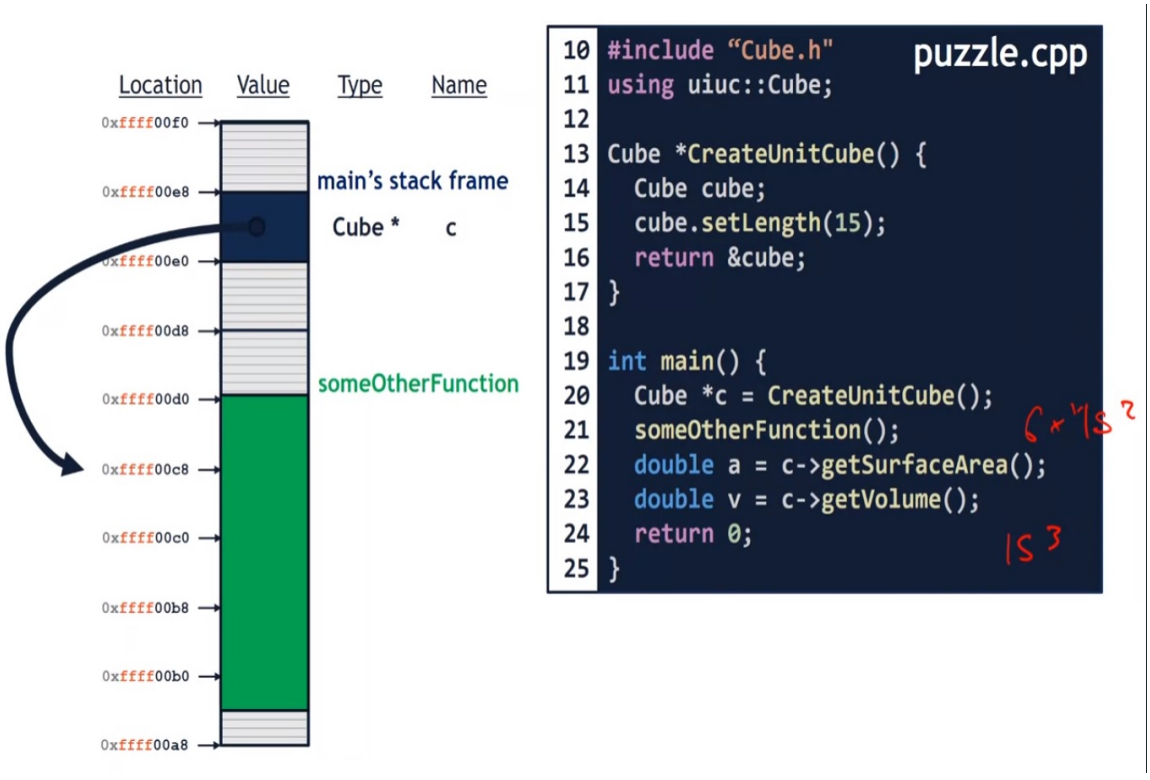

Figure 1.2: Illustration of how Stack memory works with pointers applied to the cube class from Listing 1.2. As this code is written, it will either produce zero or a segmentation fault. This is because we are returning a reference to a local variable.

1.2.3 Heap Memory

If memory needs to exist for longer than the life cycle of the function, we must use Heap Memory.

Definition: Program Lifecycle - The lifecycle is the stages a computer program undergoes from its initial creation to deployment and execution. The stage are: edit time, compile time, link time, distribution time, installation time, load time, and run time.

Basically if the memory allocation a function needs to exist for longer than that functions lifecycle, we need to define it as Heap memory.

Definition: Heap Memory - the portion of memory where dynamically allocated memory resides.

- The only way to create heap memory in C++ is with the ”new” operator.

C++’s new operator

The ”new” operator returns a pointer to the memory storing the data, not an instance of the data itself. The ”new” operator in C++ will always do three things:

1. Allocate memory on the heap for the data structure.
2. Initialize the data structure.
3. Return a pointer to the start of the data structure.

The memory is only ever reclaimed by the system when the pointer is passed to the ”delete” operator. These concepts are explicitly explained in Listing 1.5 with C++. An illustration along with an explanation of how this memory is being allocated and assigned is given in Figure 1.3.

Listing 1.5: Heap memory walkthrough in C++.

```
//The code below allocates two chunks of memory:
//Memory to store an integer pointer on the stack.
//Memory to store an integer on the heap.

int * numPtr = new int;

//Here we allocate our numPtr pointer to stack memory through "int * numPtr".
//The value of this pointer is going to be a "new int"
//This means we are pointing the value of our pointer
//to an integer in heap memory.
```

```
//This heap memory will run throughout the entire lifecycle of the program
//unless we call delete.

#include <iostream>

int main() {
  int *numPtr = new int;

  std::cout << "*numPtr: " << *numPtr << std::endl;
  //We haven't put any memory on the heap, so it will print out a random integer.
  //It depends on what was in that memory before it was accessed.
  std::cout << " numPtr: " <<  numPtr << std::endl;
  //This is the contents of numPtr itself, which is the address of Heap memory.
  //Heap memory always starts at the lowest value and goes up.
  //This will spit out a significantly lower value
  //than the stack memory address.
  std::cout << "&numPtr: " << &numPtr << std::endl;
  //The address of numPtr, will be a large address since it is
  //an address on the stack.

  *numPtr = 42;
  //We set the derferenced pointer to 42.
  std::cout << "*numPtr assigned 42." << std::endl;

  std::cout << "*numPtr: " << *numPtr << std::endl;
  //Now this will output 42, since we are setting *numPtr to point to 42.
  std::cout << " numPtr: " <<  numPtr << std::endl;
  //The address of the Heap memory will remain the same.
  std::cout << "&numPtr: " << &numPtr << std::endl;
  //The address of the Stack memory will remain the same.

  return 0;
}
```

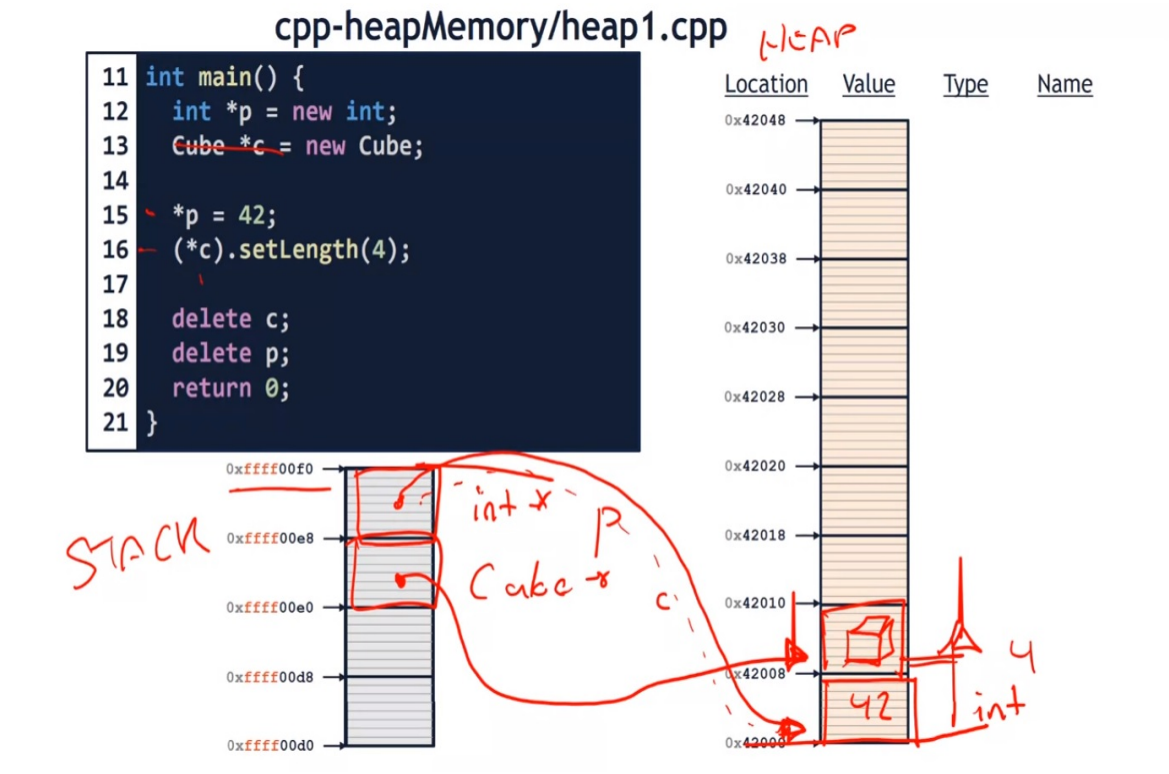

Figure 1.3: Illustration of how heap and stack memory work through the cube class from Listing 1.2. We can tell memory is in stack by its relatively large hexadecimal value compared to heap memory. Heap memory will start at small values and goes up, while stack memory starts at large values and goes down. We allocate ”int *p” and ”Cube *c” to the stack memory and point them to separate values on the heap. So now we have two stack variables and two heap variables. Next we say the dereferenced value of p is set to 42, and the dereferenced value of c’s length is set to 4. After that we delete c. When we delete c, we look at the variable c, we see what memory it’s pointing to, and we go ahead and delete this memory, giving to back to the system. Likewise we delete p afterwards as the stored integer and give it back to the system. However, there is something still left here, we have the stack memory. The stack memory will get deleted when main returns but we also have these pointers that point to memory that doesn’t actually contain any data for us. The solution to this is with a Null Pointer. We can fix this by adding ”c = nullptr;” and ”p = nullptr” after we delete it. This is generally good practice to do as well!

Null Pointer in C++: nullptr

The C++ keyword ”nullptr” is a pointer that points to the memory address 0x0.

- nullptr represents a pointer to ”nowhere”.
- Address 0x0 is reserved and never used by the system.
- Address 0x0 will always generate a ”segmentation fault” when accessed.
- Calls to ”delete” 0x0 are ignored.

Arrow Operator (- >)

When an object is stored via a pointer, access can be made to member functions using the arrow operator. The call “(*c).setLength(4);” is equivalent to “c− >setLength(4);”

1.2.4 Heap Memory Puzzles

The goal of this section is to practice our understanding of heap/stack memory allocation.

Listing 1.6: Puzzle 1: pointers and stack memory in C++

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
  int i = 2, j = 4, k = 8;
  //First we set the values of i,j,k.
  int *p = &i, *q = &j, *r = &k;
  //Next we declare integer pointer p,q,r to point to i,j,k respectively.

  k = i;
  //Here we overwrite the value of k with i.
  //So the value of k is 2, and the address of k is the address of i.
  cout << i << j << k << *p << *q << *r << endl;
  //We print out i, j, k, dereference p, dereference q, dereference r.
  //This will spit out 242242
  //This is because the value of k is the value of i,
  //and the pointer *r is pointing to the address of k,
```

```
  //which is the address of i.

  p = q;
  //p takes in the value of q, so p no longer points to i,
  //rather it points to the same place as q which is the address of j.
  cout << i << j << k << *p << *q << *r << endl;
  //This will spit out 242442

  *q = *r;
  //Here we set the dereferenced value of q
  //to be the dereferenced value of r.
  //The dereferenced value of q, is j.
  //The dereferenced value r, is k.
  //So the value of j takes on 2.
  cout << i << j << k << *p << *q << *r << endl;
  //This will spit out 222222

  return 0;
}
```

Listing 1.7: Puzzle 2: pointers and heap memory in C++

```
/**
 * C++ puzzle program.  Try to figure out the result before running!
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include <iostream>

using std::cout;
using std::endl;

int main() {
  int *x = new int;
  //We set x to be an integer pointer in stack memory.
  //It points to a new integer in heap memory.
```

```
  int &y = *x;
  //int &y is giving us a reference variable.
  //A reference variable aliases another piece of memory
  //which allows us to give a name to a piece of memory.
  //What this means is, we are calling the heap memory address of x as y.
  y = 4;
  //Then we set the value in the heap memory address, to be 4.

  cout << &x << endl;
  //This should be a large hexadecimal number since it is stack memory.
  cout << x << endl;
  //The contents of x should be a small hexadecimal number since
  //its value is a pointer that points to a heap memory address.
  cout << *x << endl;
  //The dereferenced contents of x should be 4 since
  //we set the dereference value of x to the variable y,
  /which is equal to 4.

  cout << &y << endl;
  //The memory address of y is the heap memory address of x.
  cout << y << endl;
  //The value of y is 4.
  // cout << *y << endl;
  //What is the dereferenced value of a non-pointer?
  //We actually can't do this which is why its commented out.
  //Because it is not a pointer, we can't dereference a non pointer.

  return 0;
}
```

Listing 1.8: Puzzle 3: pointers and heap/stack memory in C++

```
/**
 * C++ puzzle program.  Try to figure out the result before running!
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */
```

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
  int *p, *q;
  //We declare two integer pointers p and q in stack memory.
  p = new int;
  //New heap memory is going to be pointed to by p.
  q = p;
  //Here we declare the contents of q to be the contents of p.
  //This means that q points to the same heap memory as p.
  *q = 8;
  //We set the pointed value (dereferenced value) of q to be 8.
  cout << *p << endl;
  //Thus the pointed value of p is going to be
  //the pointed value of q which is 8.

  q = new int;
  //Now we set q to point to a different heap memory than p.
  *q = 9;
  //The dereferenced value for q is 9.
  cout << *p << endl;
  //Since we haven't changed what the old heap memory points to,
  //p will still point to same value of 8.
  cout << *q << endl;
  //The value q points to is 9.

  return 0;
}
```

Listing 1.9: Puzzle 4: pointers and heap/stack memory in C++

```
/**
 * C++ puzzle program.  Try to figure out the result before running!
 *
```

```
 * @author
 *    Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include <iostream>

using std::cout;
using std::endl;

int main() {
  int *x;
  //We declare x to be an integer pointer in stack memory.
  int size = 3;
  //size is an integer of value 3.
  x = new int[size];
  //x points to an array of integers with size 3 in heap memory.
  //Each value in the array takes on its own heap memory address.

  for (int i = 0; i < size; i++) {
    x[i] = i + 3;
    //The value of each element in heap memory is set to ith element plus 3.
  }

  delete[] x;
  //We have to delete with brackets to remove all the contents.
}
```

1.2.5 Assignment for Memory

Listing 1.10: Test assignment showing how to allocate a member of a class into heap memory for C++.

```
#include <iostream>

// This class Pair has already been defined for you.
// (You may not change this definition.)
class Pair {
public:
  int first, second;
```

```
  void check() {
    first = 5;
    std::cout << "Congratulations! The check() method of the Pair class
    \n has executed. (But, this isn't enough to guarantee
    \n that your code is correct.)" << std::endl;
  }
  Pair *pairFactory();
};

// Insert your declaration and implementation of the function pairFactory()
// below, by replacing "..." with proper C++ code. Be sure to declare the
// function type to return a pointer to a Pair.

Pair *pairFactory() {
  return new Pair;
}

// Your function should be able to satisfy the tests below. You should try
// some other things to convince yourself. If you have a bug in this problem,
// the usual symptom is that after you submit, the grader will crash with a
// system error. :-)
int main() {
  Pair *p = pairFactory();

  // This function call should work without crashing:
  p->check();

  // Deallocating the heap memory. (Assuming it was made on the heap!)
  delete p;

  std::cout << "If you can see this text, the
  system hasn't crashed yet!" << std::endl;

  return 0;
}
```

1.3 Developing C++ classes

1.3.1 Constructors

Whenever an instance of a class is created, the class constructor sets up the initial state of the object of interest. If there is not a specific constructor, then C++ will assign an automatic constructor. It should be noted that the automatic default constructor only initializes member variables to their default values.

Custom Default Constructor

Unlike an automatic default constructor which happens when no constructors are declared, the custom default constructor specifies the state of the object when the object is constructed. We can define one in the following way:

- A member function with the same name of the class (ex: "Cube::Cube()).
- The function takes zero parameters.
- The function does not have a return type.

There is code written in C++ demonstrating how the custom default constructor is implemented in Listing 1.11

Listing 1.11: A custom default constructor presented in C++

```
/**
 * Simple C++ class for representing a Cube (with a custom constructor).
 *
 * @author
 *    Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#pragma once

namespace uiuc {
  class Cube {
    public:
      Cube();  // Custom default constructor

      double getVolume();
```

```
      double getSurfaceArea();
      void setLength(double length);

    private:
      double length_;
  };
}

//Cube.cpp

#include "Cube.h"

namespace uiuc {
  Cube::Cube() {
    length_ = 1;
  }

  double Cube::getVolume() {
    return length_ * length_ * length_;
  }

  double Cube::getSurfaceArea() {
    return 6 * length_ * length_;
  }

  void Cube::setLength(double length) {
    length_ = length;
  }
}

//main.cpp

#include "Cube.h"
#include <iostream>

int main() {
  uiuc::Cube c;
  std::cout << "Volume: " << c.getVolume() << std::endl;
```

```
  return 0;
}
```

Custom Constructors

We can also specify custom, non-default constructors that require client code (any code that imports our class and makes use of the data type in the class) to supply arguments. An example would be in the form of

Cube::Cube(double length)// one-argument ctor specifying initial length.

Constructors are not unique meaning you can have custom default constructors and multiple constructors. It should be noted that if any custom constructor is defined, an automatic default constructor is not defined.

Listing 1.12: A custom default constructor presented in C++

```
/**
 * Simple C++ class for representing a Cube (with constructors).
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#pragma once

namespace uiuc {
  class Cube {
    public:
      Cube(); // Custom default constructor
      Cube(double length); // One argument constructor

      double getVolume();
      double getSurfaceArea();
      void setLength(double length);

    private:
      double length_;
  };
```

```
}

//Cube.cpp

#include "Cube.h"

namespace uiuc {
  Cube::Cube() {//Custom default constructor
    length_ = 1;
  }

  Cube::Cube(double length) {//Custom one argument constructor
    length_ = length;
  }

  double Cube::getVolume() {
    return length_ * length_ * length_;
  }

  double Cube::getSurfaceArea() {
    return 6 * length_ * length_;
  }

  void Cube::setLength(double length) {
    length_ = length;
  }
}

//main.cpp

#include "Cube.h"
#include <iostream>

int main() {
  uiuc::Cube c(2);
  //When we call c(2), C++ will take the constructor which takes in
  //the single value, not the default constructor.
  std::cout << "Volume: " << c.getVolume() << std::endl;
```

```
    return 0;
}
```

1.3.2 Copy Constructors

In C++, a copy constructor is a special type of constructor that allows us to make a copy of an existing object.

Definition: Automatic Copy Constructor-Just like before, the automatic copy constructor will be provided for our class for free from the C++ compiler.

Custom Copy Constructor

A custom copy constructor is:

- A class constructor.
- Has exactly on argument:
  - The argument must be a const reference of the same type as the class.
  - Constant reference or reference to a constant basically means that you can't modify the value of type object to which the reference refers.
- Syntax: "Cube::Cube(const Cube & bbj)"
  - Here the implementation of the parameter spells as, we are entering a constant (const) cube (Cube) passed by reference (& bbj).

A trivial example of how this is applied is given in Listing 1.13.

Often times, a copy constructor is invoked automatically:

- Passing an object as a parameter (by value).
- Returning an object from a function (by value),
- Initializing a new object.

Examples of how copy constructors are used in code can be found in Listing 1.13

Listing 1.13: Custom copy constructor presented in C++ through examples

```
#include "Cube.h"
#include <iostream>
```

```
namespace uiuc {
  Cube::Cube() {//Custom default constructor
    length_ = 1;
    std::cout <<"Default constructor invoked!" << std::endl;
  }

  //In this case the automatic copy constructor would do the same thing.
  Cube::Cube(const Cube & obj) {//Custom copy constructor
  //This copies the content of our object and copies
  //it into our current instance.
    length_ = obj.length_;
    std::cout <<"Copy constructor invoked!" << std::endl;
  }
  //As we define data structures, we will need to use copy constructors.

  ...

//cpp-cctor/ex1/main.cpp

#include"../Cube.h"
using uiuc::Cube;

void foo(Cube cube){

}

int main(){
    Cube c;
    //Creating the cube will invoke the default constructor.
    foo(c);
    //This is a call to the copy constructor.

    return 0;

}

//cpp-cctor/ex2/main.cpp
```

```
#include"../Cube.h"
using uiuc::Cube;

void foo(Cube cube){
    Cube c;
    //This cube will call the default constructor.
    return c;
    //We have an object in foo(), and we need to copy it into main()
    //so that main() can make use of it.
    //This will call the copy constructor.
}

int main(){
    Cube c2 = foo();
    //c2 takes on the return value of foo().

    //Once it gets back into mains stack frame, we need to do something with this
    //In order to do this, it needs to invoke a copy constructor.

    //We expect to see 1 default constructor called,
    //and 2 copy constructors invoked after.

    return 0;

}

//cpp-cctor/ex3/main.cpp

#include"../Cube.h"
using uiuc::Cube;

int main(){
    Cube c;
    //This will call a default constructor.
    Cube myCube=c;
    //Since we are declaring myCube as c,
    //we are copying the contents of c into myCube.
    //This will call a copy constructor.
```

```
    return 0;

}

//cpp-cctor/ex4/main.cpp

#include"../Cube.h"
using uiuc::Cube;

int main(){
    Cube c;
    //This will call a default constructor.
    Cube myCube;
    //This will call a default constructor.

    myCube = c;
    //This will not call a copy constructor.
    //The reason why is because we are not doing any construction.
    //The objects myCube and c are already constructed above.
    //This line just assigns the contents of myCube to be the contents of c
    //which are the same by default.

    return 0;

}
```

1.3.3 Copy Assignment Operators

A copy assignment defines the behaviour when an object is copied using the assignment operator (=). In contrast to a copy constructor which creates a new object (constructor) an assignment operator assigns a value to an existing object. Whenever an object is constructed, an assignment operator is always called.

Automatic Assignment Operator

When no assignment operator is provided, the C++ compiler will provide one automatically.

Definition: Automatic Assignment Operator - The automatic assignment operator will copy the contents of all member variables.

Custom Assignment Operator

The custom assignment operator is:

- Is a public member function of the class.
- Has the function name "operator=".
- Has a return value of a reference of the class' type.
- Has exactly one argument
  - The argument must be const reference of the class' type.
- Syntax: Cube & Cube:: operator=(const Cube & obj)
  - This reads as: The custom assignment operator which belongs to the Cube class (Cube::operator=) has a return value of a Cube by reference (Cube &), and it passes a single argument of a constant reference of the class' type (const Cube & obj).
  - The goal of the custom assignment operator is to assign the contents in "obj" to the intance of the class that is being called upon (Cube &).

An example of this is provided in Listing 1.14.

Listing 1.14: Custom assignment operator implemented in C++ through the class structure

```
/**
 * Simple C++ class for representing a Cube (with constructors).
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#pragma once

namespace uiuc {
  class Cube {
    public:
      Cube();  // Custom default constructor
      Cube(const Cube & obj);  // Custom copy constructor
```

```
    Cube & operator=(const Cube & obj);  // Custom assignment operator

    double getVolume();
    double getSurfaceArea();
    void setLength(double length);

  private:
    double length_;
 };
}

//Cube.cpp

#include "Cube.h"
#include <iostream>

namespace uiuc {
 Cube::Cube() {
  length_ = 1;
  std::cout << "Default constructor invoked!" << std::endl;
 }

 Cube::Cube(const Cube & obj) {
  length_ = obj.length_;
  std::cout << "Copy constructor invoked!" << std::endl;
 }

 Cube & Cube::operator=(const Cube & obj) {
  //The function name is operator=.
  //It returns a Cube by reference.
  //The argument is a constant Cube by reference.
  length_ = obj.length_;
  std::cout << "Assignment operator invoked!" << std::endl;
  return *this;
  //We will always return a dereferenced value of "this".
  //"this" denotes the instance of the class itself in C++
  //in this case (Cube &).
 }
```

```
  double Cube::getVolume() {
    return length_ * length_ * length_;
  }

  double Cube::getSurfaceArea() {
    return 6 * length_ * length_;
  }

  void Cube::setLength(double length) {
    length_ = length;
  }
}

//main.cpp

#include "Cube.h"
using uiuc::Cube;

int main() {
  Cube c;
  //Invokes default constructor.
  Cube myCube;
  //Invokes default constructor.

  myCube = c;
  //myCube and c have already been constructed.
  //Since these are already constructed,
  //they will take on an assignment operation.

  //The functionality of a copy constructor and an assignment operation
  //are largely the same.
  //Their job is to copy the contents of one instance of a class to another.
  //The invocation of an assignment operator means the object already exists
  //and doesn't need construction.

  return 0;
}
```

1.4 Variable Storage

In C++, an instance of a variable can be stored directly in memory, accessed by a pointer, or accessed by reference. An illustration of this for the Cube class is shown in Figure 1.4.

1.4.1 Direct Storage

By default, variables are stored directly in memory.

- The type of a variable has no modifiers.
- The object takes up exactly its size in memory.
- Ex:
  - Cube c; //Stores a Cube in memory
  - int i; //Stores an integer in memory
  - uiuc::HSLAPixel p;//Stores a pixel in memory

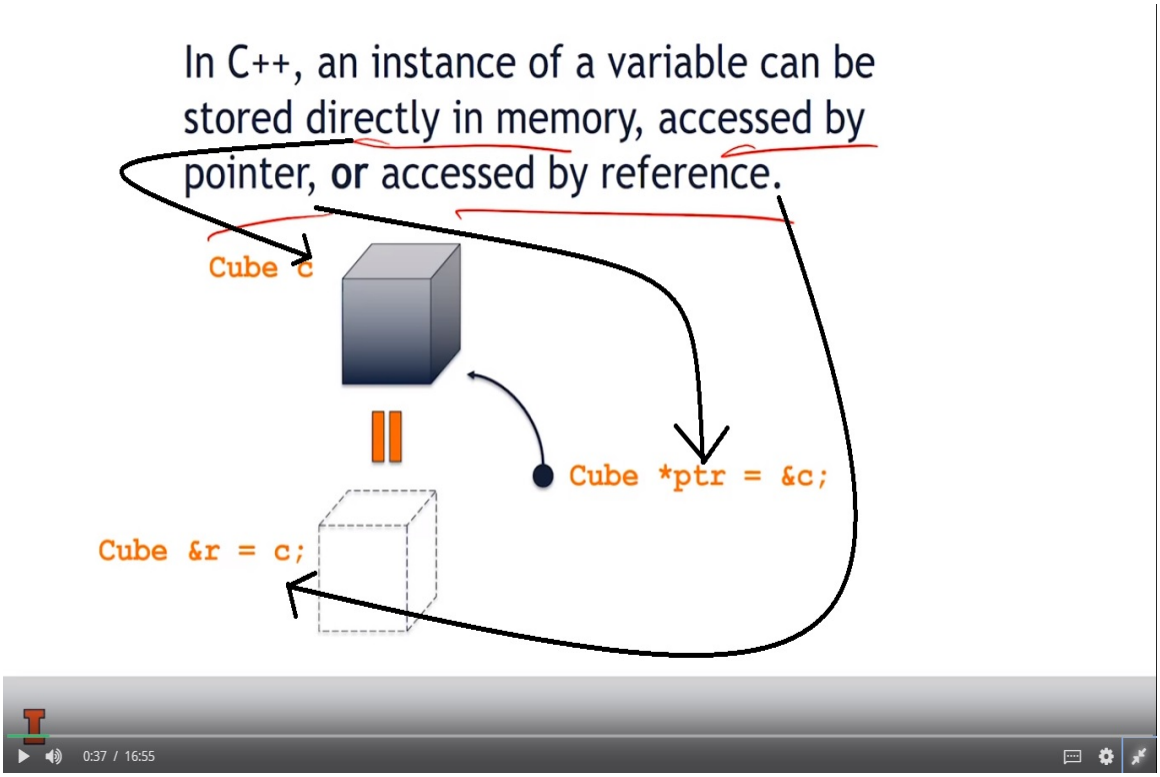

Figure 1.4: Illustration showing how variables are stored in C++

1.4.2 Storage by Pointer

- The type of a variable is modified by the dereference operator (*).
- A pointer takes a "memory address width" of memory (ex: 64 bits on a 64-bit system).
- The pointer "points" to the allocated space of the object.
- Ex:
  - Cube *c; //Pointer to a Cube in memory
  - int *i; //Pointer to an integer in memory
  - uiuc::HSLAPixel *p;//Pointer to a pixel in memory

1.4.3 Storage by Reference

- A reference is an alias to existing memory and is denoted in the type with an ampersand (&)
- A reference does not store memory itself, it is only an alias to another variable.
- The alias must be assigned when the variable is initialized.
- Ex:
  - Cube &c = cube; //Alias to the variable 'cube'
  - int &i = count; //Alias to the variable 'count'
  - uiuc::HSLAPixel &p; //This declaration is illegal since it has to be assigned when it is declared (it is not creating an alias for 'p').

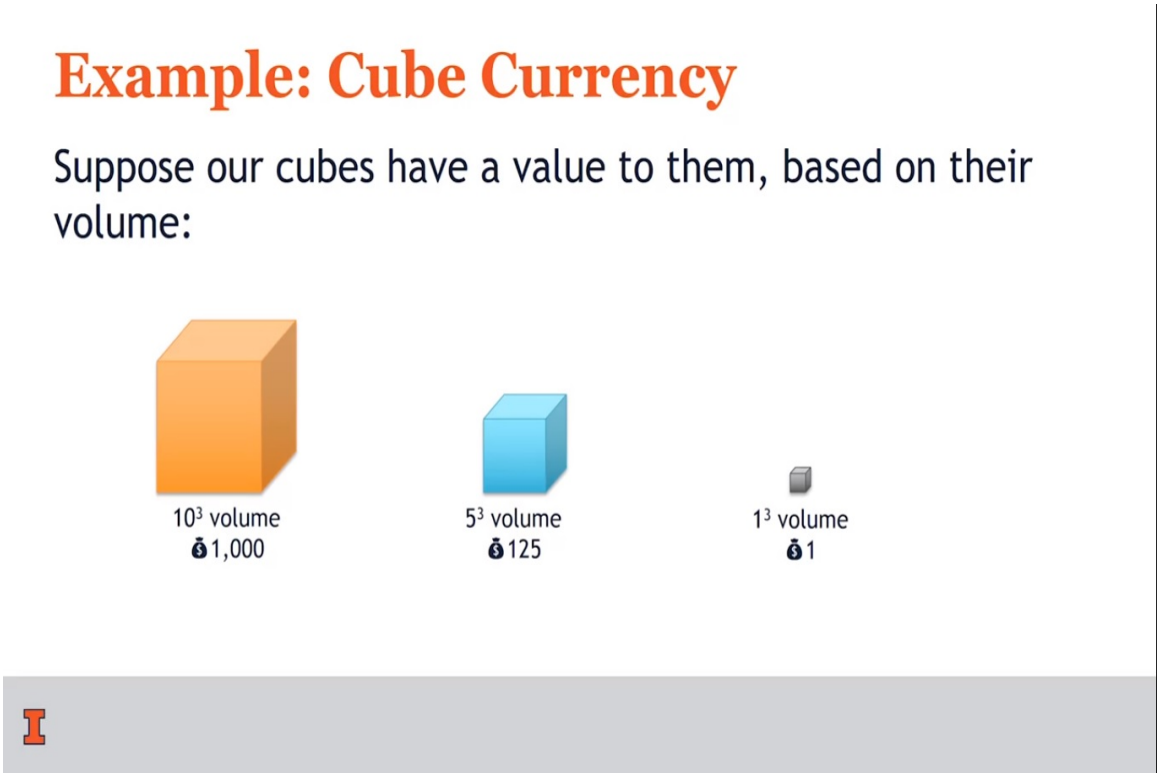

Figure 1.5: Visualizing the example we will work through. The larger the length size the more valuable the cube is in some generic currency. When we receive money, we want the cube itself, not a copy of the cube. All we want to do is transfer the cube to someone else and understand what is happening to the cube in memory.

1.4.4 Example Applied to Variable Storage

The example is visualized and explained in Figure 1.5. The example is worked through with the different variable storages' in Listing 1.15.

Listing 1.15: Worked through example of utilizing memory for a class as described in Figure 1.5

```
/**
 * Simple C++ class for representing a Cube (with constructors).
 *
 * @author
 *    Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

//Cube.h
```

```
#pragma once

namespace uiuc {
  class Cube {
    public:
      Cube(double length);  // One argument constructor
      Cube(const Cube & obj);  // Custom copy constructor

      Cube & operator=(const Cube & obj);  // Custom assignment operator

      double getVolume() const;
      double getSurfaceArea() const;
      void setLength(double length);

    private:
      double length_;
  };
}

//cpp-memory2/Cube.cpp

#include "Cube.h"
#include <iostream>

namespace uiuc {
  Cube::Cube(double length) {
    //This is a one parameter constructor.
    length_ = length;
    std::cout << "Created $" << getVolume() << std::endl;
  }
  //This constructor creates money (creates a new Cube).


  Cube::Cube(const Cube & obj) {
    //This is a custom copy constructor.

    //This constructor takes in a Cube by reference.
```

```
    //This means that we are not going to create a new Cube
    //when we pass this argument in.

    //Instead we are going to have the cube aliased to us.
    //This aliased cube is going to copy into the newly constructed cube.

    length_ = obj.length_;
    std::cout << "Created $" << getVolume() << " via copy" << std::endl;
}
//This constructor creates new money (creates a new class)
//by copying its content.

//We want to be very careful when new money is created.

Cube & Cube::operator=(const Cube & obj) {
    //This is an assignment operator.

    //This assignment operator transforms an instance from one value to another.
    std::cout << "Transformed $" << getVolume() << "-> $" << obj.getVolume()
    std::cout << std::endl;
    //It is going to transform its old volume (getVolume()) into
    //a new volume (obj.getVolume()).
    length_ = obj.length_;
    return *this;
}

//All three of these constructors are creating money.
//From a computer science standpoint, whenever the constructor
//is called it is taking up memory.

double Cube::getVolume() const {
    return length_ * length_ * length_;
}

double Cube::getSurfaceArea() const {
    return 6 * length_ * length_;
}
```

```
  void Cube::setLength(double length) {
    length_ = length;
  }
}

//cpp-memory2/ex1/byValue.cpp
//What happens when we transfer a cube to
//another cube by value?
//All we want to do is transfer $1000 from one person
//to another.

#include "../Cube.h"
using uiuc::Cube;

int main() {
  // Create a 1,000-valued cube
  Cube c(10);

  // Transfer the cube
  Cube myCube = c;
  //We create our first cube and then we copy our second cube
  //thus, creating a total of $2000.

  return 0;
}

//cpp-memory2/ex1/byRef.cpp
//What happens when we transfer a cube to
//another cube by reference?
//All we want to do is transfer $1000 from one person
//to another.

#include "../Cube.h"
using uiuc::Cube;

int main() {
  // Create a 1,000-valued cube
  Cube c(10);
```

```
  // Transfer the cube
  Cube & myCube = c;
  //We create our first cube and then we alias myCube as c.
  //Note: we are not creating any new memory when we do this.
  //Thus, we are creating a total of $1000.

  //However, we haven't actually transferred the money,
  //we just aliased two people to owning the same cube.

  return 0;
}

//cpp-memory2/ex1/byPointer.cpp
//What happens when we transfer a cube to
//another cube by pointer?
//All we want to do is transfer $1000 from one person
//to another.

#include "../Cube.h"
using uiuc::Cube;

int main() {
  // Create a 1,000-valued cube
  Cube c(10);

  // Transfer the cube
  Cube * myCube = &c;
  //We create the first cube, the we create a Cube pointer
  //called myCube and it points to the address memory of c.

  //Instead of aliasing two values to the same thing,
  //we are creating the thousand dollars and then
  //pointing the other variable to that $1000

  return 0;
}
```

1.4.5 Pass by _____

Using the example illustrated in Figure 1.5, lets try to get something more complicated and more akin to actually transferring money, instead of just changing variables. What we actually care about is how we transfer the money from one person to another. The best way to do this is by passing variables through functions.

Identical to storage, arguments can be passed to functions in three different ways:

- Pass by value (default).
- Pass by pointer (modified with *) by creating a pointer that we pass along.
- Pass by reference (modified with &, acts as an alias) which doesn't make a copy, it just makes two variables alias the same memory.

A walk through about how the different pass by arguments are implemented is shown in Listing 1.16.

Listing 1.16: Worked through example of passing arguments through functions by using the class described in Figure 1.5

```
/**
 * C++ program sending a Cube by value.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

//Using the same header file and constructors as before.
//cpp-memory2/ex2/byValue.cpp
//Passing the cube by value.
#include "../Cube.h"
using uiuc::Cube;

bool sendCube(Cube c) {
  //By passing the c by value, we are copying the content
  //into the argument when its called in the main program.
  // ... logic to send a Cube somewhere ...
  return true;
}
```

```
int main() {
  // Create a 1,000-valued cube
  Cube c(10);

  // Send the cube to someone
  sendCube(c);
  //Here we are passing the Cube by value into sendCube
  //In doing so we copy the content of c into sendCube.

  //This will create $1000, and then copy an additional $1000.

  return 0;
}

//Using the same header file and constructors as before.
//cpp-memory2/ex2/byRef.cpp
//Passing the cube by reference.
#include "../Cube.h"
using uiuc::Cube;

bool sendCube(Cube &c) {
  //By passing c by reference, we are not copying the content
  //into the argument since no copies are made!
  // ... logic to send a Cube somewhere ...
  return true;
}

int main() {
  // Create a 1,000-valued cube
  Cube c(10);

  // Send the cube to someone
  sendCube(c);
  //Here we are passing the Cube by reference into sendCube.
  //In doing so we are sending an alias of this variable
  //meaning nothing is created or copied.

  //This will create $1000.
```

```
  return 0;
}

//Using the same header file and constructors as before.
//cpp-memory2/ex2/byPointer.cpp
//Passing the cube by Pointer.
#include "../Cube.h"
using uiuc::Cube;

bool sendCube(Cube *c) {
  //By passing the Cube pointer *c, we are not copying the content
  //into the argument since no copies are made!
  // ... logic to send a Cube somewhere ...
  return true;
}

int main() {
  // Create a 1,000-valued cube
  Cube c(10);

  // Send the cube to someone
  sendCube(&c);
  //Here we are passing the memory address of c into sendCube.
  //In doing so we are creating a Cube pointer *c which
  //points to value at that specific memory address.
  //This means the $1000 is only created once, and the argument
  //is pointing to that value.

  //This will create $1000.

  return 0;
}
```

1.4.6 Return by _____

Similarly, values can be returned in all three ways as:

- Return by value (default) which makes a copy.
- Return by pointer (modified with *) we are returning a memory address associated with that variable.
- Return by reference (modified with &, acts as an alias) which returns an alias to that variable.
  - Never return a reference to a stack variable created on the stack of your current function.

1.5 Class Destructor

When an instance of a class is cleaned up, the class destructor is the last call in a class's life cycle.

1.5.1 Automatic Default Destructor

If there are no other destructors defined, an automatic default destructor is added to your class. This calls the default destructor of all member objects. Destructors are called during run time not at compile time and they should never be called directly. Instead, they are automatically called when the object's memory is being reclaimed by the system:

- If the object is on the stack, when the function returns.
- If the object is on the heap, when "delete" is called is used.

1.5.2 Custom Destructor

We can add custom behaviour to the end-of-life of the function through a custom destructor by:

- A custom destructor is a member function.
- The function's destructor is the name of the class, preceded by a tilde $\sim$.
- All destructor have zero arguments and no return type.
- Syntax: "Cube::$\sim$ Cube() //Custom destructor"

A custom destructor is essential when an object allocates an external resource that must be closed or freed when the object is destroyed. Examples:

- Heap memory,
- open files,
- shared memory.

A custom destructor is implemented in the Listing 1.17.

Listing 1.17: Worked through example of destructors by using the class described in Figure 1.5. The ~ in the code isn't portrayed accurately due to lstlistings environment.

```
/**
 * Simple C++ class for representing a Cube (with destructors).
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

//cpp-dtor/Cube.h

#pragma once

namespace uiuc {
  class Cube {
    public:
      Cube();  // Custom default constructor
      Cube(double length);  // One argument constructor
      Cube(const Cube & obj);  // Custom copy constructor
      ~Cube();  // Destructor

      Cube & operator=(const Cube & obj);  // Custom assignment operator

      double getVolume() const;
      double getSurfaceArea() const;
      void setLength(double length);

    private:
      double length_;
```

```
  };
}

//cpp-dtor/Cube.cpp

#include "Cube.h"
#include <iostream>

using std::cout;
using std::endl;

namespace uiuc {
  Cube::Cube() {
    length_ = 1;
    cout << "Created $1 (default)" << endl;
  }

  Cube::Cube(double length) {
    length_ = length;
    cout << "Created $" << getVolume() << endl;
  }

  Cube::Cube(const Cube & obj) {
    length_ = obj.length_;
    cout << "Created $" << getVolume() << " via copy" << endl;
  }

  Cube::~Cube() {
    cout << "Destroyed $" << getVolume() << endl;
  }

  Cube & Cube::operator=(const Cube & obj) {
    cout << "Transformed $" << getVolume() << "-> $" << obj.getVolume() << endl;
    length_ = obj.length_;
    return *this;
  }
```

```
  double Cube::getVolume() const {
    return length_ * length_ * length_;
  }

  double Cube::getSurfaceArea() const {
    return 6 * length_ * length_;
  }

  void Cube::setLength(double length) {
    length_ = length;
  }
}

//cpp-dtor/main.cpp

#include "Cube.h"
using uiuc::Cube;

double cube_on_stack() {
  Cube c(3);
  //Constructs a cube of length 3.
  return c.getVolume();
  //Because it is on the stack, it is going to get destroyed
  //as soon as we call the return value.
  //From the dsetructor, we will destroy the cube of length 3.
}

void cube_on_heap() {
  Cube * c1 = new Cube(10);
  //We create a pointer cube *c1 that points to memory on heap
  //that is assigned a value of a cube with length 10.
  Cube * c2 = new Cube;
  //We create a pointer cube *c2 that points to memory on heap
  //that is assigned a value of a cube an automatic value of length 1.
  delete c1;
  //We destroy the cube c1 of length 10, but we do not destroy
  //the cube c2 of length 1.
```

```
}

int main() {
  cube_on_stack();
  //This will create a cube of volume $27, then it will
  //destroy a cube of volume $27$
  cue_on_heap();
  //This will create a cube of volume $1000, and a cube of volume $1.
  //Then it will destroy the cube of volume $1000.
  cube_on_stack();
  //This will create a cube of volume $27, then it will
  //destroy a cube of volume $27$

  //We expect seven lines to appear.
  return 0;
}
```

1.5.3 Practice assignment and solution

A class called Pair has already been declared, but the constructors have not been implemented yet. Pair has two public member variables:

int *pa,*pb;

These two ”pointers to int” are intended to point to heap memory locations that store integers. The remainder of the Pair class expects the following functionality.

- A single constructor Pair(int a, int b): This should set up pa and pb to point to newly allocated memory locations on the heap. The integers at those memory locations should be assigned values according to the constructor's integer arguments a and b.

- A copy constructor Pair(const Pair & other): This takes as its argument a read-only reference to another Pair. It should set up the newly constructed Pair as a ”deep copy,” that is, it should create a new Pair that is equivalent to the other Pair based on dereferenced values but does not reuse any of the same memory locations. In other words, the copy constructor should set up its own instance's member variables pa and pb to point to newly allocated memory locations for integers on the heap; those memory locations must be new, not the same locations pointed to by the other Pair, but the integers at these new locations should be assigned values according to the integers that the other Pair is pointing to.

- A destructor Pair() that de-allocates all of the the heap memory that had previously been allocated for this Pair's members.

Listing 1.18: Worked through example incorporating all the concepts on class making.

```
/* Class Pair has already been declared
 * as shown in the following comments:
 *
 * class Pair {
 * public:
 *   int *pa,*pb;
 *   Pair(int, int);
 *   Pair(const Pair &);
 *  ~Pair();
 * };
 *
 * Implement its member functions below.
 */

Pair::Pair(int a, int b){
  pa = new int;
  //We set pa to point to heap memory.
  *pa = a;
  //The value it points to is a.
  pb = new int;
  //We set pb to point to heap memory.
  *pb = b;
   //The value it points to is b.

}

Pair::Pair(const Pair & P){
  //This copy constructor creates a new Pair that is equivalent
  //to the custom constructor above, except it does not reuse the same
  //memory location.

  pa = new int;
  //We set pa to point to a different heap memory.
```

```
    *pa = *(P.pa);
    //The copied pa will point to the given pairs pa.
    pb = new int;
    //We set pb to point to a different heap memory.
    *pb = *(P.pb);
     //The copied pb will point to the given pairs pb.

 }

Pair::~Pair(){
    delete pa;
    delete pb;
    //Destruct pa and pb from the heap memory.
 }


 /* Here is a main() function you can use
  * to check your implementation of the
  * class Pair member functions.
  */

int main() {
  Pair p(15,16);
  //We call the custom constructor and assign pa to point to 15
  //and pb to point to 16 in heap.
  Pair q(p);
  //We copy the content of p into q and allocate pa and pb
  //to point to different locations in heap.
  Pair *hp = new Pair(23,42);
  //We create a Pair pointer call *hp to point to heap memory
  //that has contents of the class Pair with a=23 and b=42.
  delete hp;

  std::cout << "If this message is printed,"
    << " at least the program hasn't crashed yet!\n"
    << "But you may want to print other diagnostic messages too." << std::endl;
  return 0;
}
```

1.6 Engineering C++ Software Solutions

1.6.1 Template Types

A template type is a special type that can take on different types when the type is initialized. An example of a template type is “std::vector”.

std::vector

The standard vector (std::vector) is a standard library class that provides the functionality of a dynamically growing array wit a ”templated” type. The arguments of the std::vector can take on any desired object such as; std::vector¡char¿, std::vector¡int¿, std::vector¡uiuc::Cube¿. Some of the key ideas behind the vector type are:

- Defined in: # include < vector >
- Initialization: std::vector < T > v //T is the desired object
  - When initializing a ”templated” type, the template type goes inside of <> at the end of the type name, i.e. std::vector <object> v;
- Add to (back) of array: push_back(T); //v.push_back(T)
- Access specific element: operator[](unsigned pos); //v[0];
- Number of elements: size();//v.size();

Listing 1.19: Using the vector template in C++

```
/**
 * C++ program using the std::vector class.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

//cpp-vector/ex1/main.cpp

#include <vector>
#include <iostream>

int main() {
  std::vector<int> v;
```

```
  v.push_back( 2 );
  v.push_back( 3 );
  v.push_back( 5 );

  std::cout << v[0] << std::endl;
  std::cout << v[1] << std::endl;
  std::cout << v[2] << std::endl;

  return 0;
}
```

```
//cpp-vector/ex2/main.cpp

#include <vector>
#include <iostream>

int main() {
  std::vector<int> v;
  for (int i = 0; i < 100; i++) {
    v.push_back( i * i );
  }

  std::cout << v[12] << std::endl;

  return 0;
}
```

1.7 Tower of Hanoi

This problem is very fundamental in computer science and will be studied extensively for this course.

1.7.1 Introduction to the problem

The Tower of Hanoi problem is quite familiar and it involves moving a stack of cubes from side to the other where the volume of the bottom cube must be larger than the volume of the cube on top of it. For a small amount of sequentially larger cubes, this problem is easy to solve visually. As more cubes are introduced, the problem becomes increasingly more

difficult and we want to create a programmed solution which will solve this easily for an arbitrary number of cubes. This problem is illustrated in Figure 1.6.

1.7.2 Building the Tower of Hanoi Game

A new class must be created to represent each of the stacks in the Tower of Hanoi game. Each stack will have an unknown number of cubes allocated to it which makes the vector template a great tool for solving this problem. We also need some sort of operations that interact with the top of the stack. Notice how we never actually care about what is below the top cube whenever we are applying an operation to it. The Stack class must contain:

- A vector of Cubes
- Operations to interact with the top of the stack.

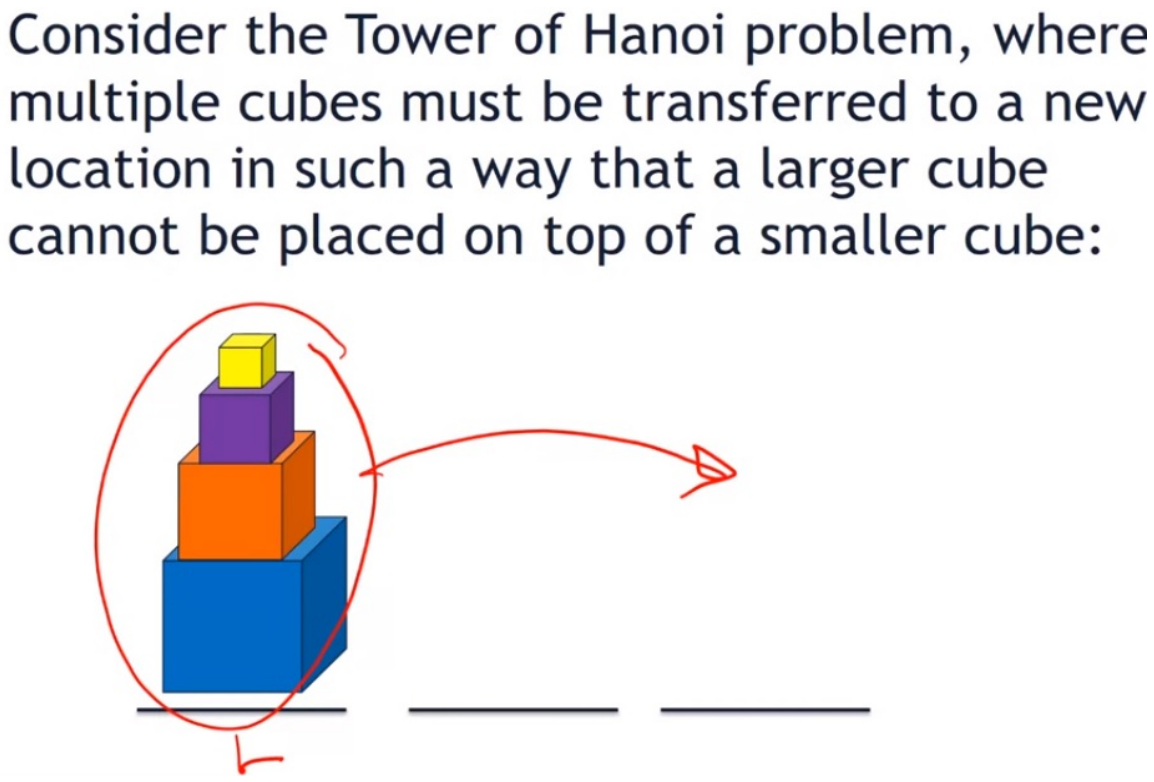

Figure 1.6: Visualizing the Tower of Hanoi problem. Each cube is categorized with a different colour, and the object of the problem is to move the stacks of blocks from one side to the other without putting any larger blocks on top of the smaller blocks. Each empty space is called a stack and it has a certain number of cubes allocated to it. In this case here, the stacks would be allocated as Stack[0], Stack[1], Stack[2], going from left to right respectively. You can only move one cube at a time!

and the game itself is built on:

- An array of three stacks
- The initial state has four cubes in the first stack.

The appropriate classes are constructed for us in Listing 1.20. Our goal is to write a function (solve()) that solves this problem for us.

Listing 1.20: Header files defining each class for the Tower of Hanoi problem.

```
/**
 * Simple C++ class for representing a Cube (with constructors).
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

//cpp-tower/uiuc/Cube.h

//The main difference between this cube and the previous one is that
//we have two private variables compared to earlier were
//each cube now has a length, and a color associated to it.

#pragma once

#include "HSLAPixel.h"

namespace uiuc {
  class Cube {
    public:
      Cube(double length, HSLAPixel color);

      double getLength() const;
      void setLength(double length);

      double getVolume() const;
      double getSurfaceArea() const;

    private:
```

```
    double length_;
    HSLAPixel color_;
  };
}

//cpp-tower/Stack.h

//This Stack class implements the operations for each stack.

#pragma once

#include <vector>
#include "uiuc/Cube.h"
using uiuc::Cube;

class Stack {
  public:
    void push_back(const Cube & cube);
    //This borrows terminology from the vector template.
    //It pushes an element to the last element of a Stack.
    Cube removeTop();
    //This returns a Cube by value, so it returns a copy
    //of a Cube created earlier.
    Cube & peekTop();
    //This returns a Cube by reference, and it only returns
    //the memory address of the Cube on top, without
    //removing that Cube from the internal data structure.
    unsigned size() const;
    //The size() function returns the size of each stack
    //in a similar manner to the vector template.

    // An overloaded operator<<, allowing us to print the stack via 'cout<<':
    friend std::ostream& operator<<(std::ostream & os, const Stack & stack);
    //A friend function of a class is defined outside that class' scope
    //but it has the right to access all private and protected
    //members of the class.

    //It allows us to cout an object.
```

```
    //It will take on the std::ostream& operator and it will take in
    //itself and a constant type of the class it is in.
  private:
    std::vector<Cube> cubes_;
};

//cpp-tower/Stack.cpp

#include "Stack.h"

#include <exception>
#include <iostream>
using std::cout;
using std::endl;

void Stack::push_back(const Cube & cube) {
  // Ensure that we do not push a cube on top of a smaller cube:
  if ( size() > 0 && cube.getLength() > peekTop().getLength() ) {
    //size() is defined to return the size of the current cube called in push_bac
    std::cerr << "A smaller cube cannot be placed on top of a larger cube." << st
    std::cerr << "  Tried to add Cube(length=" << cube.getLength() << ")" << std
    std::cerr << "  Current stack: " << *this << std::endl;

    throw std::runtime_error("A smaller cube cannot be placed on top of a larger
    // A C++ exception is a response to an exceptional circumstance that
    //arises while a program is running, such as an attempt to divide by zero.

    //The throw keyword throws an exception in the program when a problem shows u
    //A throw expression accepts one parameter (in this case have a runtime error
    //which is passed as an argument to the exception handler.

    //In this case, the exception is the conditional.
    //The conditional is that the size of the current cube is positive and that t
    //length is greater than size of the previous cube on the stack (i.e. peekTop

    //A handle can be anything from an integer index to a pointer to a resource i
    //The idea is that they provide an abstraction of a resource,
```

```
    //so you don't need to know much about the resource itself to use it.

    //Here we are creating a handler within the push_back function that will make
    //we are properly stacking the cubes for each Stack[] placement in Game.
    //If the condition isn't met, the exception isn't called, and we simply push
    //the current vector of Cubes.
  }

  cubes_.push_back(cube);
}

Cube Stack::removeTop() {
  Cube cube = peekTop();
  //We declare cube to be a Cube class that has the address of the top cube on th
  //vector of Cubes cubes_
  cubes_.pop_back();
  //Removes the last element in the vector, effectively reducing the container si
  //Opposite of push_back()
  return cube;

  //Essentially, we are defining an object that is a Cube which has an address gi
  //the top cube of the current stack, reducing that current stacks size by 1, an
  //returning the result of that.
}

Cube & Stack::peekTop() {
  //Accesses the memory address of the top cube, without removing it from the Sta
  return cubes_[cubes_.size() - 1];
}

unsigned Stack::size() const {
  return cubes_.size();
}

std::ostream& operator<<(std::ostream & os, const Stack & stack) {
  for (unsigned i = 0; i < stack.size(); i++) {
    os << stack.cubes_[i].getLength() << " ";
  }
```

```
  os << endl;
  return os;
}

//cpp-tower/Game.h

#pragma once

#include "Stack.h"
//We will use Stacks from Stack.h
#include <vector>

class Game {
  public:
    Game();
    //Game constructor will set up the initial state of the class.
    void solve();
    //The solve() function will solve the game for us.

    // An overloaded operator<<, allowing us to print the stack via 'cout<<':
    friend std::ostream& operator<<(std::ostream & os, const Game & game);

  private:
    std::vector<Stack> stacks_;
    //Takes in Stack as the template for a private vector
    //called stacks_
};

//cpp-tower/Game.cpp

#include "Game.h"
#include "Stack.h"
#include "uiuc/Cube.h"
#include "uiuc/HSLAPixel.h"

#include <iostream>
using std::cout;
using std::endl;
```

```
// Solves the Tower of Hanoi puzzle.
// (Feel free to call "helper functions" to help you solve the puzzle.)
void Game::solve() {
  // Prints out the state of the game:
  cout << *this << endl;

  // @TODO -- Finish solving the game!
}

// Default constructor to create the initial state:
Game::Game() {
  // Create the three empty stacks:
  for (int i = 0; i < 3; i++) {
    Stack stackOfCubes;
    stacks_.push_back( stackOfCubes );
    //stackOfCubes will be added to the back of
    //stacks_ array.
  }

  // Create the four cubes, placing each on the [0]th stack:
  Cube blue(4, uiuc::HSLAPixel::BLUE);
  stacks_[0].push_back(blue);

  Cube orange(3, uiuc::HSLAPixel::ORANGE);
  stacks_[0].push_back(orange);

  Cube purple(2, uiuc::HSLAPixel::PURPLE);
  stacks_[0].push_back(purple);

  Cube yellow(1, uiuc::HSLAPixel::YELLOW);
  stacks_[0].push_back(yellow);
}

std::ostream& operator<<(std::ostream & os, const Game & game) {
  for (unsigned i = 0; i < game.stacks_.size(); i++) {
    os << "Stack[" << i << "]: " << game.stacks_[i];
  }
```

```
  return os;
}

//cpp-tower/main.cpp

#include "Game.h"
#include <iostream>

//The main function that is solving the Tower of Hanoi
//game

int main() {
  Game g;

  std::cout << "Initial game state: " << std::endl;
  std::cout << g << std::endl;

  g.solve();

  std::cout << "Final game state: " << std::endl;
  std::cout << g << std::endl;

  return 0;
}
```

Listing 1.21: Solution file for the Tower of Hanoi problem.

```
//My solution for the Tower of Hanoi class given

//game.cpp

#include "Game.h"
#include "Stack.h"
#include "uiuc/Cube.h"
#include "uiuc/HSLAPixel.h"

#include <iostream>
using std::cout;
```

```
using std::endl;

// Solves the Tower of Hanoi puzzle.
// (Feel free to call "helper functions" to help you solve the puzzle.)

//Algorithm
//1. Make the legal move between Stack[0], and Stack[1] (in either direction)
//2. Make the legal move between Stack[0], and Stack[2] (in either direction)
//3. Make the legal move between Stack[1], and Stack[2] (in either direction)
//Repeat

void Game::Step1(){

  Stack s0=stacks_[0];
  Stack s1=stacks_[1];

  Cube dflt(0, uiuc::HSLAPixel::DEFAULT);
  //I defined a default cube so that I could
  //set a value of zero for an empty stack.

  if(s1.size()<1){
    stacks_[1].push_back(dflt);
  }
  if(s0.size()<1){
    stacks_[0].push_back(dflt);
  }
  //If a stack has no cubes, allocate a cube of length zero (dftl).
  Cube & s1top=stacks_[1].peekTop();
  Cube & s0top=stacks_[0].peekTop();

  if(s0top.getLength() > s1top.getLength() && s1top.getLength()==0){
    stacks_[1].removeTop();
    stacks_[1].push_back(s0top);
    stacks_[0].removeTop();
    return;
  }

  if(s0top.getLength() < s1top.getLength() && s0top.getLength()==0){
```

```
    stacks_[0].removeTop();
    stacks_[0].push_back(s1top);
    stacks_[1].removeTop();
    return;
  }

  if(s0top.getLength() < s1top.getLength() && s1top.getLength()!=0){
    stacks_[1].push_back(s0top);
    stacks_[0].removeTop();
    return;
  } else {
    stacks_[0].push_back(s1top);
    stacks_[1].removeTop();
    return;
  }

}

void Game::Step2(){

  Stack s0=stacks_[0];
  Stack s2=stacks_[2];

  Cube & s0top=s0.peekTop();

  Cube dflt(0, uiuc::HSLAPixel::DEFAULT);

  if(s2.size()<1){
    stacks_[2].push_back(dflt);
  }
  Cube & s2top=stacks_[2].peekTop();

  if(s0top.getLength() > s2top.getLength() && s2top.getLength()==0){
    stacks_[2].removeTop();
    stacks_[2].push_back(s0top);
    stacks_[0].removeTop();
    return;
  }
```

```
  if(s0top.getLength() < s2top.getLength() && s2top.getLength()!=0){
    stacks_[2].push_back(s0top);
    stacks_[0].removeTop();
    return;
  } else {
    stacks_[0].push_back(s2top);
    stacks_[2].removeTop();
    return;
  }

}

void Game::Step3(){

  Stack s1=stacks_[1];
  Stack s2=stacks_[2];

  Cube & s1top=s1.peekTop();
  Cube & s2top=stacks_[2].peekTop();


  if(s1top.getLength() < s2top.getLength() && s2top.getLength()!=0){
    stacks_[2].push_back(s1top);
    stacks_[1].removeTop();
    return;
  } else {
    stacks_[1].push_back(s2top);
    stacks_[2].removeTop();
    return;
  }

}

void Game::solve() {

  if(stacks_[2].size()<3){
    Step1();
```

```
    cout << "Step1"<< endl;
    cout << *this << endl;
    Step2();
    cout << "Step2"<< endl;
    cout << *this << endl;
    Step3();
    cout << "Step3"<< endl;
    cout << *this << endl;
    solve();
  }

}

// Default constructor to create the initial state:
Game::Game() {
  // Create the three empty stacks:
  for (int i = 0; i < 3; i++) {
    Stack stackOfCubes;
    stacks_.push_back( stackOfCubes );
  }

  Cube blue(4, uiuc::HSLAPixel::BLUE);
  stacks_[0].push_back(blue);

  Cube orange(3, uiuc::HSLAPixel::ORANGE);
  stacks_[0].push_back(orange);

  Cube purple(2, uiuc::HSLAPixel::PURPLE);
  stacks_[0].push_back(purple);

  Cube yellow(1, uiuc::HSLAPixel::YELLOW);
  stacks_[0].push_back(yellow);
}

std::ostream& operator<<(std::ostream & os, const Game & game) {
  for (unsigned i = 0; i < game.stacks_.size(); i++) {
    os << "Stack[" << i << "]: " << game.stacks_[i];
  }
```

```
  return os;
}
```

1.7.3 Templates and Classes

C++ allows for us to use templates and classes. A templated variable is defined by declaring it before the beginning of a class or function. The format is shown in Listing 1.22. Templated variables are checked at compile time, which allows for errors to be caught before running the program:

Listing 1.22: Template syntax in C++.

```
//Implementing a template variable
template <typename T>
class List{

    private:
    T data_;
    //Can have a list of any object since the template type is initialized

}

//Implementing a template function

template <typename T>
int max(T a, T b){
    if (a >b) {return a;}
    return b
}

/**
 * Simple use of C++ templates.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#include <iostream>
#include <string>
```

```
using std::cout;
using std::endl;

#include "Cube.h"
using uiuc::Cube;

// We'll call this my_max to avoid conflicts with the "max" in the
// standard libraries.
template <typename T>
T my_max(T a, T b) {
  if (a > b) { return a; }
  return b;
}

int main() {
  cout << "my_max(3, 5): " << my_max(3, 5) << endl;
  cout << "my_max('a', 'd'): " << my_max('a', 'd') << endl;

  // Here we construct std::string objects from the literal strings in
  // quotation marks, because the std::string object already implements
  // the ">" operator to do alphabetical ordering. A plain string literal
  // is an array of const char, which wouldn't support that correctly.
  // (Instead, it would just compare the addresses of the arrays.)
  cout << "my_max(std::string(\"Hello\"), std::string(\"World\")): "
       << my_max(std::string("Hello"), std::string("World")) << endl;

  // You need to finish implementing the ">" operator for Cube to get the
  // next line to work!
  // cout << "my_max( Cube(3), Cube(6) ): " << my_max( Cube(3), Cube(6) )
  << endl;

  return 0;
}
```

1.7.4 Inheritance

Inheritance allows for a class to inherit all member functions and data from a base class into a derived class. A base class is a generic form of a specialized, derived class. The syntax for these concepts is shown in Listing 1.23.

When a base class is inherited, the derived class has access to all the public members of the base class but the derived class can not access private members of the base class.

Listing 1.23: Inheritance syntax in C++.

```
/**
 * Generic 'Shape' class.
 *
 * @author
 *   Wade Fagen-Ulmschneider <waf@illinois.edu>
 */

#pragma once

class Shape {
  public:
    Shape();
    //Constructor
    Shape(double width);
    //One parameter constructor
    double getWidth() const;

  private:
    double width_;
};

namespace uiuc {
  class Cube : public Shape {
  //We declare that Shape is the base class by declaring it as public after
  //the derived class.
    public:
      Cube(double width, uiuc::HSLAPixel color);
      //Notice that width doesn't need to be declared as a private
      //variable since it is inherited from the Shape class.
```

```
    double getVolume() const;

    //99% of inheritance is used in the public context.

  private:
    uiuc::HSLAPixel color_;
  };
}

//cpp-inhertiance/Cube.cpp

#include "Cube.h"
#include "Shape.h"

namespace uiuc {
  Cube::Cube(double width, uiuc::HSLAPixel color) : Shape(width) {
    color_ = color;
  }

  double Cube::getVolume() const {
    // Cannot access Shape::width_ due to it being 'private'
    // ...instead we use the public Shape::getWidth(), a public function

    return getWidth() * getWidth() * getWidth();
    //Since width is the private member of another class, we can't access any of
    //the private members that have been inherited.
  }
}

//cpp-inhertiance/Cube.cpp

//Initialized list

#include "Shape.h"

Shape::Shape() : Shape(1) {
  // Nothing.
}
```

```
Shape::Shape(double width) : width_(width) {
  // Nothing.
}

double Shape::getWidth() const {
  return width_;
}
```

Initialization

When a derived class is initialized, the derived class must construct the base class:

- Cube must construct Shape.
- By default, uses default constructor.
- Custom constructor can be used with an initialization list.

Initializer List

The syntax to initialize the base class is called the initializer list and can be used to:

- Initialize a base class.
- Initialize the current class using another constructor.
- Initialize the default values of member variables.

Chapter 2

Programming in Java

Java is a popular and widely used language. It is the predominant language for android devices. It is also fairly accessible to beginners.

2.1 Programming Syntax and Basics with Java

In general, much of the syntax between C++ and Java are identical. This is largely because Java was developed from C++ and takes many of the core principles behind C++ and expands upon it. Some of the key advantages of Java compared to C++ are, garbage collection, the build process, simplicity in source code, a binary standard, dynamic linking, portability, and a standard type system as examples. Although there are still some distinct advantages for C++ such as startup time for small code, the amount of memory used, garbage collection pausing, and allowing for custom deterministic destructors.

2.1.1 Compiling and Executing Java Programs

Java is an object-oriented programming language. All code in java is organized in classes and the methods contained in the class are written in source code with "name".java files. The compilation process takes source code in java and turns it into bytecode files which have the extension "name".class. The command for compiling the java file into a bytecode file is "javac name.java"

Unlike C++, java doesn't create separate object and executable files. Instead both of those files are put into a bytecode file which you can execute on the command line with the command "java name". This reduces the complexity when building programs and drastically simplifies the source code. Think about going from header files, makefiles, cpp files, main file to just one java file!

Some important things to note are that:

- Eery function must be part of a class.
- Every class is part of a package.
- A public class can be used in any package.
- A non-public class can only be used in its own package.

The command line approach for compiling a simple java program in a bash & Linux environment would be

User:∼/directory $ javac name.java

User:∼/directory $ java name

2.1.2 Writing Programs within Java

In Java, everything is written within a class. This means all functions which are called methods, all declarations, variables, objects, and even the main program is encapsulated within a specific class. The main program is the start of all classes (as similar to C++) and it has a special declaration in the form of "public static void main(String []args)".

Listing 2.1: Writing the first program in Java.

```
//To compile a java program, you type in the command
//'javac "name".java' on the command line
//Unlike C++, java doesn't create a separate
//executable that you run.
//Instead it create a bytecode file which contains the
//the object file and executable file in machine code.
//To run the bytecode you type 'java ''name"'


public class hello {
//Class containing the hello world program.

  public static void main(String []args) {
  //The main method.
  //The main method takes in arguments through the command line
  //which is similar to C++.
  //The static command
```

```
    System.out.println("Hello, World!");
    //Print our "Hello, World!" and end the line.
    //.print("") prints without ending the line.

    System.out.println("The sum of 2 and 3 is 5.");

    int sum = Integer.parseInt(args[0]) + Integer.parseInt(args[1]);
    //'type'.parse'type' converts from a string to the desired type.
    //This is done here so that math can be done.

    System.out.format("The sum of %s and %s is %s.\n",
      args[0], args[1], Integer.toString(sum));
  }
  //Return back to string to print out final statement.
}

//The + operator is useful for printing and it is overloaded to work
//on strings as follows.
//If either operand is a string, it: converts the other operand to a String,
//or it creates a new String by concatenating both operands.
//This means that combining strings and numerics often leads to concatenation
//rather than arithmetic.
```

2.1.3 Garbage Collection Java vs. C++

One of the main strengths with Java is that it uses automatic garbage collection. This means that Heap memory does not need to be manually deleted after declaration as it does in C++. As such, writing complicated programs in C++ can often lead to memory shortages due to mismanaged memory which can lead to catastrophic failure of the dynamic memory management system.

In fact, with Java, all objects are dynamically allocated on Heap and with automatic garbage collection, Java naturally deals with this memory in an easier manner. Dynamic memory allocation is useful because it allows programs to run with less memory and take up less resources than needed. This makes Java excellent for automatically optimizing large programs because objects are only stored in memory as needed. In contrast to static memory which can take up many resources that it doesn't need to take up. Due to this nature, Java can naturally utilize multithreading without communicating directly to the compiler because multiple parts of the program can run simultaneously from the automatic dynamic

Primitive Types	
Type	Description
boolean	True, false, same as C++
char	Holds one character
byte	8-bit signed integer
short	16-bit signed integer
int	32-bit signed integer
long	64-bit signed integer
float	floating-point nunber
double	double precision floating-point number
Reference Types	
arrays	These are really pointers
classes	These are really pointers

Table 2.1: Table describing the different varible types supported by Java . There is no struct, union, enum, unsigned, or typdef types and arrays & classes are pointers.

memory allocation.

This can also lead to an overuse in memory that isn't necessary in Java. This is particularly relevant for small programs which don't have much memory allocation. It also means that you have more control over how your program accesses memory in C++ which is why it is the predominant language used for explicit parallel programming.

2.1.4 Java Types

In Java, there are two "categories" of types: primitive types and reference types. These are summarized in Table 2.1.

2.1.5 C++ Arrays vs. Java Arrays

In C++, the standard declaration of an array leads to storage in Stack memory. In Java however, the declaration of an array is really only declaring a pointer to an array. The storage of the array is not allocated until you use the "new" command which is similar to C++ for allocating to Heap memory, except without the nuance in tracking pointers/references. Some coding examples are shown in Listing 2.2.

Listing 2.2: Arrays in Java vs. C++.

```
//C++
```

```
int A[10];
A[0] = 5;

//Java

int [] A; // A is a pointer to an array.
A = new int [10]; // Now A points to an array of length 10.
A[0] = 5; // Set the 1st elements value to 5 of the array pointed to by A.

int [] A = {1,222,0};
//A points to an array of length 3
//containing the values 1, 222, 0

//Uninitalized values are given defaults of
//boolean -> false
//char -> "\u0000"
//byte,int,short,long,float,double -> 0
//any pointer -> null

//Out of bounds array indexes always create a runtime error.

//You can determine the length of an array using the ".length" method.

A = new int[20];
A.length;//This evaluates to 20.

//In Java, arrays can be copied using the arraycopy function.
//System.arraycopy(src,srcPos,dst,dstpos,count)
//src=source
//dst=destination
//count=how many values to copy.

int [] A,B;
A = new int[10];
//code to put values in A
B = new int[5];
System.arraycopy(A,0,B,0,5);
//This copies the first 5 elements of A into B.
```

```
//arraycopy will create runtime errors if the size of the arrays is
//improperly referenced or elements are chosen out of bounds.

//arraycopy also works on the same array.

int [] A={0,1,2,3,4};
System.arraycopy(A,0,A,1,4);

//Java arrays can be multidimensional.

int [][] A;
A = new int[5][];
A[0] = new int [1];
A[1] = new int [2];
//rest allocated to whatever
```

2.1.6 Java classes vs C++ classes

Variables declared whose type is a class are allocated for an object of that class in C++. In Java, you are really declaring a pointer to a class object meaning no storage is allocated for the class object and any constructors are only called when storage is allocated through the "new" command. The context in which this is shown is given by Listing 2.3.

This can lead to aliasing problems in Java since every object is a pointer and references aren't explicitly declared. Also in Java, all values are passed by value, but for arrays and classes, the actual parameter is really pointer. This means that if we change the array element or a class field inside of a function, this will change the actual parameters element or field. These are shown and explained in Listing 2.4.

Listing 2.3: Classes in Java vs. C++.

```
class List {
  public void AddToEnd(...)
  { ...}
  ...
}
//            C++

List L;          // L is a List; the List constructor function is called to
```

```
                    // initialize L.
List *p;            // p is a pointer to a List;
                    // no list object exists yet, no constructor function has
                    // been called
p = new List;       // now storage for a List has been allocated
                    // and the constructor function has been called
L.AddToEnd(...)     // call L's AddToEnd function
p->AddToEnd(...)    // call the AddToEnd function of the List pointed to by p



//          JAVA

List L;             // L is a pointer to a List; no List object exists yet
L = new List();     // now storage for a List has been allocated
                    // and the constructor function has been called;
                    // note that you must use parentheses even when you are not
                    // passing any arguments to the constructor function
L.AddToEnd(...)     // no -> operator in Java -- just use .
```

Listing 2.4: Aliasing problems in Java

```
//Demonstrating the issue of aliasing between two arrays.
int [] A = new int[3];
int [] B = new int[2];

A[0] = 5;
B=A;

B[0] = 2;
//Because of this declaration, A[0] changes since the we are manually
//setting the zeroth element of B to point to a value of 2.
//Inherently, this means that the zeroth element of A
//will also point to a value of 2.

//Consequences of all parameters being passed by value.

void f( int [] A )
```

```
{
    A[0] = 10;    // change an element of parameter A
    A = null;     // change A itself
}

void g()
{
    int [] B = new int [3];
    B[0] = 5;
    f(B);
    //B is not null here, because B itself was passed by value
    //however, B[0] is now 10, because function f
    //changed the first element of the array
}

//In C++, this problem is addressed through the use of copy constructors
//which can be defined to make copies of the values pointed to,
//rather than just making copiers of the pointers.
//In Java, the solution is to use the arraycopy operation,
//or to use a class's clone operation.
```

2.1.7 Type conversions in Java

Java is much more limited in conversions of primitive types compared to C++. Booleans for example can not be converted other types. The other primitives listed in Table 2.1 have two types of conversions.

Implicit Conversions: The value of one the is changed to a value of another type without special directives from the programmer. The general rule is that implicit conversions can be done from one type to another if the range of values of the first type is a subset of the range of values of a second type. As an example, and integer can be implicitly converted to a double since it requires less memory. However a double can't be implicitly converted an integer because there is more memory needed.

- Example:

  int k=2;

  double d=k;

  This would be allowed in Java.

Explicit Conversion: Explicit conversions are done via casting. Casting can be used to convert among the primitive types except for booleans. Information can be lost due to values being truncated or doubles being rounded to integers.

- Example:

  double d=5.6;

  int k=(int)d;

  This would be allowed in Java but it would produce a value of 5.