Namespaces in Header Files

As an aside, you'll notice that the header files did not use the standard namespace. It's generally recommended to avoid using namespaces in a header file. This can help avoid naming conflicts later as functions and classes are reused in different parts of a code base.

ifndef

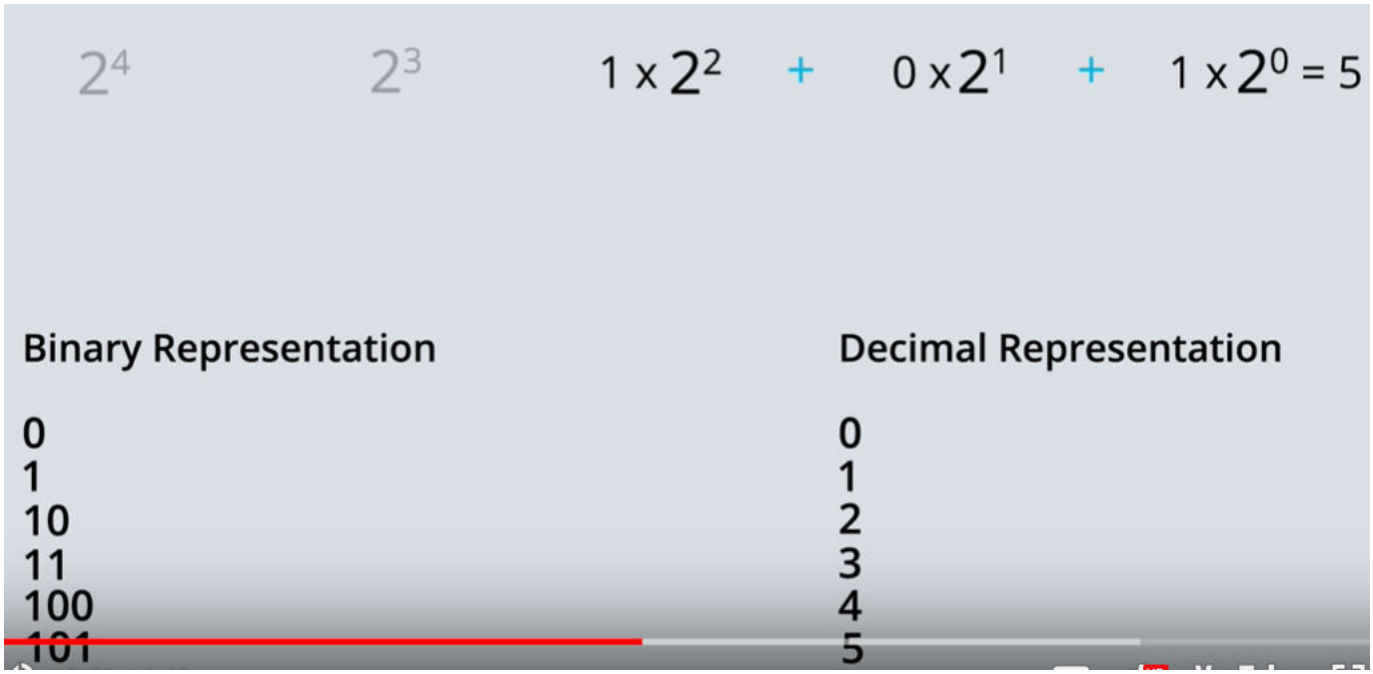
The solution is to use `# ifndef` statements, which allow you to implement a technique called inclusion guards.
The ifndef statement stands for "if not defined". When you wrap your header files with #ifndef statements, the compiler will only include a header file if the file has not yet been defined. In the current main.cpp example, the "engine.h" file would be included first.

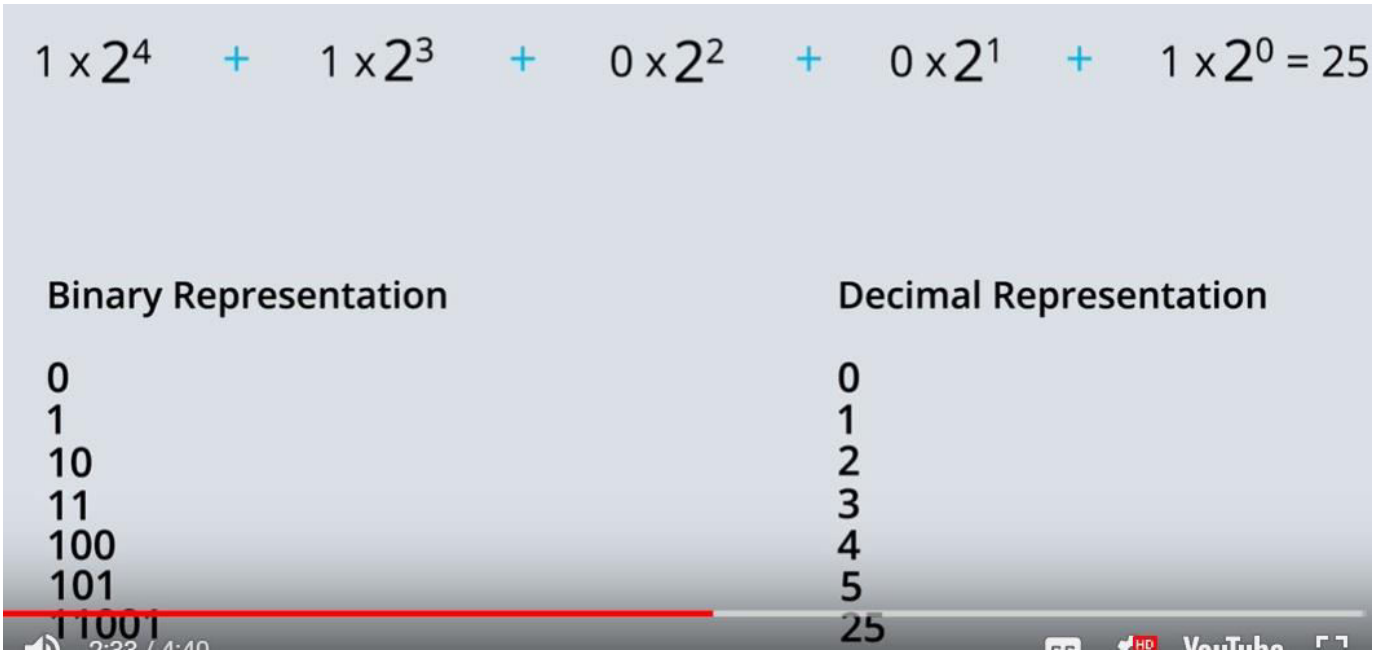
```
#ifndef FILENAME_H
#define FILENAME_H

header code ...

#endif /* FILENAME_H */
```

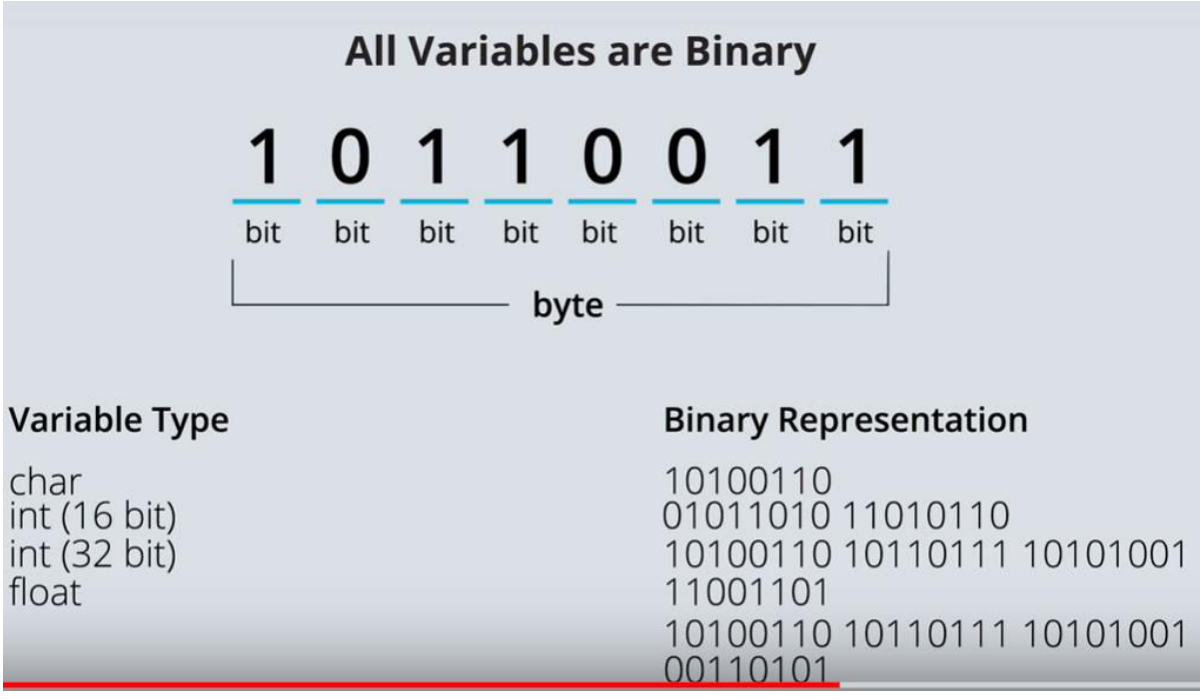
In order to storage a size_type variable:

```
std::vector<int>::size_type variablename;
```

```
#include <vector>

class Matrix
{

        private:

            std::vector< std::vector<float> > grid;
            std::vector<float>::size_type rows;
            std::vector<float>::size_type cols;

        public:

            // constructor function declarations
            Matrix ();
            Matrix (std::vector< std::vector<float> >);

            // set and get function declarations
            void setGrid(std::vector< std::vector<float> >);

            std::vector< std::vector<float> > getGrid();
            std::vector<float>::size_type getRows();
            std::vector<float>::size_type getCols();

            // matrix function declarations
            std::vector< std::vector<float> > matrix_transpose();
            std::vector< std::vector<float> > matrix_addition(Matrix);
            void matrix_print();
};
```


2^4 2^3 $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$
Binary Representation
0
1
10
11
100
101
Decimal Representation
0
1
2
3
4
5


$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 25$
Binary Representation
0
1
10
11
100
101
11001
Decimal Representation
0
1
2
3
4
5
25


All Variables are Binary
1 0 1 1 0 0 1 1
bit bit bit bit bit bit bit bit
byte
Variable Type
char
int (16 bit)
int (32 bit)
float
Binary Representation
10100110
01011010 11010110
10100110 10110111 10101001
11001101
10100110 10110111 10101001
00110101

QUESTION 1 OF 2

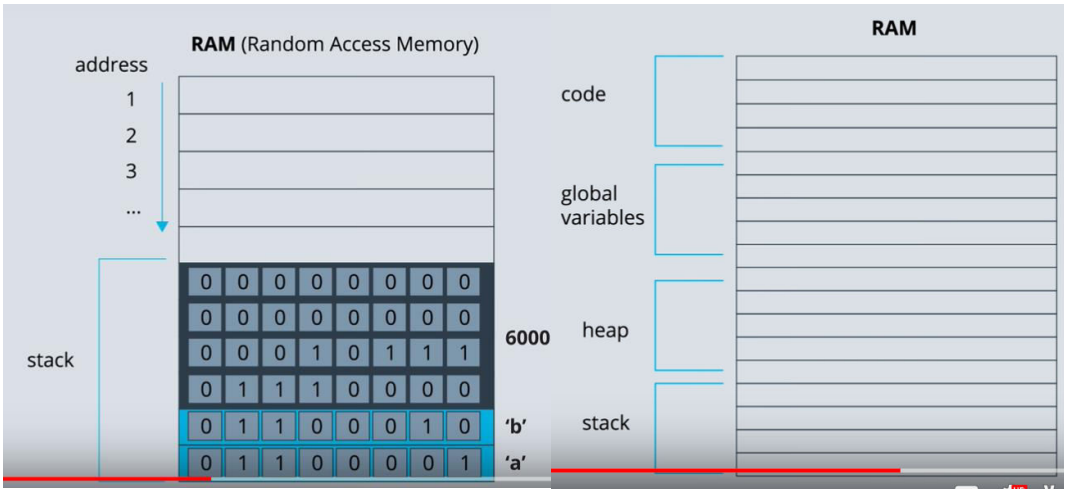
Which of the following is the 8-bit binary representation of the decimal number 63?

- 00011111
- ✓ 00111111
- 111111
- 11111100

QUESTION 2 OF 2

What integer is represented by the 16-bit binary number: 00100100 10110110

- 11510
- 37592
- 25783
- ✓ 9398



QUIZ QUESTION

When should you use dynamic memory allocation over static memory allocation?

- [ ] If you want the compiler to manage memory allocation and deallocation for you
- [x] If your variable uses more memory than what is available in the stack
- [ ] If you want to increase the speed of your programs
- [x] If you want to share variables across different functions or parts of a program
- [ ] Whenever you want to use pointers

You could break down the code development process into the following steps:

- code design
- implementing the design
- testing for bugs and fixing the bugs
- optimization

If you were optimizing a large amount of code, you would want to use something called a profiler. A profiler is a piece of software that measures how long parts of your code are taking to execute or how many resources the code uses. The profiler helps you find congestion points so that you can optimize the least efficient parts of the code first.

Testing your Code with Standard Clock

In the following exercises, we've set up some simple profiling code for you; you'll time how long it takes to run a function using the C++ standard clock.

You will optimize by:

- seeing how long it takes to run a function
- change some aspect of the code
- run the code again to see if the code runs faster

The profiling code has already been set up for you, and it looks like this:

```
#include <ctime>

std::clock_t start;
double duration;

start = std::clock();

function_name(var1, var2);

duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;

std::cout << "duration milliseconds initialize beliefs " << 1000 * duration << '\n';
```

When you declare and define a vector, the compiler reserves space in memory plus some extra bytes in case the vector expands. Once the vector's length expands past the reserved memory, the entire vector will get copied over to a different place in RAM with enough available space.

That is very inefficient! In Andy's histogram filter code, you already know how large your vectors need to be because the robot world has a fixed number of grid spaces. If you reserve space for the vector, then you can avoid all of the extra memory reallocations as you expand the vector's length. The syntax is as simple as coding:

```
std::vector<int> foo;
foo.reserve(15);
```

Now the foo vector is guaranteed to have enough space for holding fifteen integers.

Passing Variables by Reference

Whenever you call a function, C++ copies any input variables into memory even if those variables are already in memory. For fundamental data types like int, char, or float, this might not be a problem.

But with variables that take up a more significant amount of space, such as vectors, the extra copying can slow down your programs.

You'll learn how to pass variables by reference instead of by value. Passing by reference tells your function to use the variable directly in memory rather than copying the entire variable to memory over again.

```
#include "initialize_matrix.hpp"

using namespace std;
vector < vector<int> > initialize_matrix(int num_rows, int num_cols, int initial_value) {

    vector < vector<int> > matrix;
    vector<int> new_row;

    for (int i = 0; i < num_rows; i++) {
        new_row.clear();
        for (int j = 0; j < num_cols; j++) {
            new_row.push_back(initial_value);
        }
        matrix.push_back(new_row);
    }

    return matrix;
}
```

```
#include "initialize_matrix_improved.hpp"

using namespace std;
vector < vector<int> > initialize_matrix_improved(int num_rows, int num_cols, int initial_value) {

    vector < vector<int> > matrix;
    vector<int> new_row;

    for (int i = 0; i < num_rows; i++)
        new_row.push_back(initial_value);

    for (int j = 0; j < num_cols; j++)
        matrix.push_back(new_row);

    return matrix;
}
```

```
vector< vector<int> > scalar_multiply(vector< vector<int> > matrix, int scalar) {

    vector< vector<int> > resultmatrix;
    vector<int> new_row;

    vector<int>::size_type num_rows = matrix.size();
    vector<int>::size_type num_cols = matrix[0].size();

    for (int i = 0; i < num_rows; i++) {
        new_row.clear();
        for (int j = 0; j < num_cols; j++) {
            new_row.push_back(matrix[i][j] * scalar);
        }
        resultmatrix.push_back(new_row);
    }

    return resultmatrix;
```

```
vector< vector<int> > scalar_multiply_improved(vector< vector<int> > matrix, int scalar) {

    // OPTIMIZATION: Instead of creating a new variable
    // called resultmatrix, update the input matrix directly
    vector<int> new_row;

    vector<int>::size_type num_rows = matrix.size();
    vector<int>::size_type num_cols = matrix[0].size();

    for (int i = 0; i < num_rows; i++) {
        for (int j = 0; j < num_cols; j++)
            matrix[i][j] = matrix[i][j] * scalar;
    }
    return matrix;
}
```

A vector is more efficient if you specify the vector's length before pushing values. You can do this with the reserve() method, which will guarantee that the vector can hold the number of elements reserved.

```
vector< vector<int> > unreserved(int rows, int cols, int initial_value) {


    vector< vector<int> > matrix;
    vector<int> new_row;

    for (int i = 0; i < rows; i++) {
        new_row.clear();
        for (int j = 0; j < cols; j++) {
            new_row.push_back(initial_value);
        }
        matrix.push_back(new_row);
    }

    return matrix;
}
```

```
vector< vector<int> > reserved(int rows, int cols, int initial_value) {

    // OPTIMIZE: use the reserve method with the matrix and new_row variables
    vector< vector<int> > matrix;
    matrix.reserve(rows);
    vector<int> new_row;
    new_row.reserve(cols);

    for (int i = 0; i < rows; i++) {
        new_row.clear();
        for (int j = 0; j < cols; j++) {
            new_row.push_back(initial_value);
        }
        matrix.push_back(new_row);
    }

    return matrix;
}
```

References

Did you know that every time you call a function, those input variables get copied into memory?

The ampersand (&) tells the compiler to pass the input variable by reference. That means inside your function, you'll be working with the original variable instead of a copy.

The Static Keyword

Sometimes you'll need to use the same variable over and over again in your functions.

When you declare and define a variable inside a C++ function, the value gets allocated to memory.

For example,

```
some_function() {
   int x = 5;
}
```

allocates space in memory for the variable x and then assigns the value five. Then, when the function finishes, the CPU will remove the x variable from RAM. That means every time you run the function, the CPU will allocate and deallocate memory for the x variable.

If, on the other hand, your code uses the static keyword, the x variable gets allocated to memory the first time the function runs. And the x variable just stays allocated in memory for the duration of the entire program. You've just saved yourself some reads and writes to RAM:

```
some_function() {
    static int x = 5;
}
```

Notice that you need to declare and define the variable simultaneously. You cannot define a variable with the static keyword without giving the variable a value.

Global Variables versus Static Variables

Static variables are actually placed in the same area of RAM as global variables. The difference is that global variables are declared outside of functions and are available anywhere in your program to any file or function. On the other hand, static variables remain in scope. So in the above example, some_function() is the only place that can access the x variable.

For Loops

There is nothing wrong with using nested for loops (ie for loops inside of for loops). Sometimes you need them when working with C++ vectors.

However, don't use them if you don't need them! There are a few places in Andy's code where he has used nested for loops that were not needed.

If you are iterating through or initializing an m by n matrix, you might be tempted to always use nested for loops like this:

```
    for (unsigned int i = 0; i < matrix.size(); i++) {
        for (unsigned int j = 0; j < matrix[0].size(); j++) {
            do something...
        }
    }
```

Iterating through the entire matrix involves m times n operations. However, depending on what you are trying to do, you might be able to get away with doing something like this:

```
    for (unsigned int i = 0; i < matrix.size(); i++) {
        do something
    }

    for (unsigned int j = 0; j < matrix[0].size(); j++) {
        do something
    }
```

This only requires m + n operations instead of m * n operations. Remember that fewer instructions for the CPU will get your code to execute faster!

```
4  vector < vector<int> > initialize_matrix(int num_rows, int num_cols, int initial_value) {
5
6      vector < vector<int> > matrix;
7      vector<int> new_row;
8
9      for (int i = 0; i < num_rows; i++) {
10         new_row.clear();
11         for (int j = 0; j < num_cols; j++) {
12             new_row.push_back(initial_value);
13         }
14         matrix.push_back(new_row);
15     }
16
17     return matrix;
18 }
```

```
4  vector < vector<int> > initialize_matrix_improved(int num_rows, int num_cols, int initial_value) {
5
6      vector < vector<int> > matrix;
7      vector<int> new_row;
8
9      for (int i = 0; i < num_rows; i++)
10         new_row.push_back(initial_value);
11
12     for (int j = 0; j < num_cols; j++)
13         matrix.push_back(new_row);
14
15     return matrix;
16 }
```

GCC Compiler Optimization

In the classroom, you have been using a compiler called GCC. You've been compiling code with the following command:

```
g++ -std=c++11 main.cpp blur.cpp initialize_beliefs.cpp move.cpp normalize.cpp print.cpp sense.cpp zeros.
```

By default, gcc will try to lower the time it takes to compile your code; in other words, gcc optimizes for compilation time.

However, gcc can also optimize for execution time to get your code to run faster. The gcc compiler includes three levels of optimization, which you can use by adding the optimization flag to your compilation command: -O1 -O2 -O3

You can read more about what each level does at this link: Optimization Flags Link

Now that you have optimized your histogram filter code, go back and try compiling the program with the level three flag. See how much the compiler can help you speed up your code.

Here is the command for compiling with level three optimization:

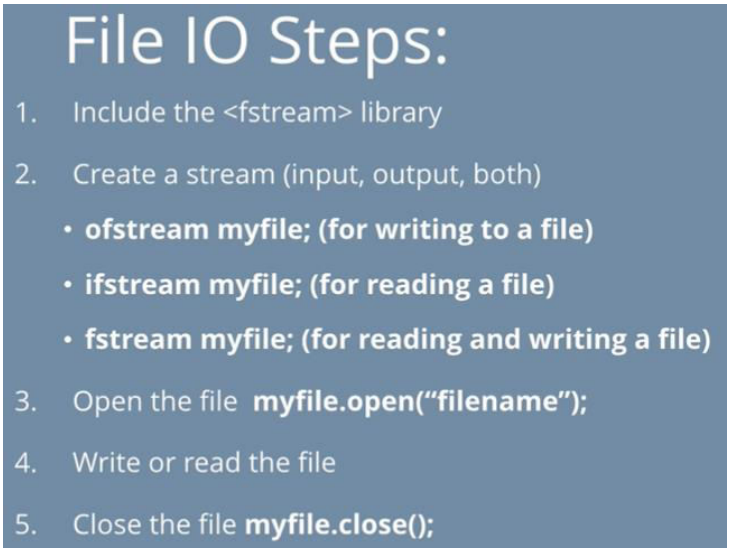
```
g++ -std=c++11 -O3 main.cpp blur.cpp initialize_beliefs.cpp move.cpp normalize.cpp print.cpp sense.cpp ze
```

```
You might have optimized using different techniques than the ones used here. The techniques used include:
- reserving memory for vectors
- passing larger variables to functions by reference
- removing variables that were not needed
- modifying vectors in place when possible instead of creating new vector variables
- iterating with ++i instead of i++
- removing dead code (lines of code that were in the files but no longer being used)
- avoiding extra for loops especially nested for loops when possible
- avoiding extra if statements
- using static and const keywords when appropriate
```

```
#include <iomanip>

std::cout<<"\n\nThe text without any formating\n";
std::cout<<"Ints"<<"Floats"<<"Doubles"<< "\n";
std::cout<<"\nThe text with setw(15)\n";
std::cout<<"Ints"<<std::setw(15)<<"Floats"<<std::setw(15)<<"Doubles"<< "\n";
std::cout<<"\n\nThe text with tabs\n";
std::cout<<"Ints\t"<<"Floats\t"<<"Doubles"<< "\n";
```

)utput will be:

```
The text without any formating
IntsFloatsDoubles

The text with setw(15)
Ints         Floats        Doubles


The text with tabs
Ints    Floats    Doubles
```




```cpp
/*Goal: Practice PreFix and PostFix
**
*/

#include<iostream>

using namespace std;

int main()
{
    int a, b = 0;
    int post, pre = 0;
    cout<<"Inital values: \t\t\tpost = "<<post<<" pre= "<<pre<<"\n";
    post = a++;
    pre = ++b;
    cout<<"After one postfix and prefix: \tpost = "<<post<<" pre= "<<pre<<"\n";
    post = a++;
    pre = ++b;
    cout<<"After two postfix and prefix: \tpost = "<<post<<" pre= "<<pre<<"\n";
    return 0;
}
```

```
Inital values:                  post = 0 pre= 0
After one postfix and prefix:   post = 0 pre= 1
After two postfix and prefix:   post = 1 pre= 2
```

```cpp
/*For this program print for each variable
**print the value of the variable,
**then print the address where it is stored.
We have a pointer and want to access the value stored in that address.
That process is called dereferencing, and it is indicated by adding the operator *
before the variable's name.
This same operator should be used to declare a variable that is meant to store a pointer.
*/
#include<iostream>
#include<string>

int main()
{
    // this is an integer variable with value = 54
    int a = 54;

    // this is a pointer that holds the address of the variable 'a'.
    // if 'a' was a float, rather than int, so should be its pointer.
    int * pointerToA = &a;

    // If we were to print pointerToA, we'd obtain the address of 'a':
    std::cout << "pointerToA points to address " << pointerToA << '\n';

    // If we want to know what is stored in this address, we can dereference pointerToA:
    std::cout << "pointerToA stores value " << * pointerToA << '\n';

    return 0;
}
```

```
pointerToA points to address 0x7fff1885b704
pointerToA stores value 54
```

```cpp
/*What is wrong with this program?*/

#include<iostream>

void increment(int input);

int main()
{
    int a = 34;
    std::cout<<"Before the function call a = "<<a<<"\n";
    increment(a);
    std::cout<<"After the function call a = "<<a<<"\n";
    return 0;
}

void increment(int input)
{
    input++;
    std::cout<<"In the function call a = "<<input<<"\n";
}

/*
So, now we know that C++ respects variable scope.
Changes to a variable that are made in a function will not effect the variable
in the main part of the program.

There are two methods to rectify this situation:
Return the altered variable
Pass the variable as a reference
*/
```

```
Before the function call a = 34
In the function call a = 35
After the function call a = 34
```

A class in C++ is a user defined data type. It can have data and functions.

The nice thing about C++ classes, the default is to make all members private.
This means only other members of the class can access the data.

That sounds bad.... we have a data type with data that we can't access. But, remember, I said C++ classes can have functions as well as data.

We can use functions to access the data in a class.

Functions that access and/or modify data values in classes are called mutators.

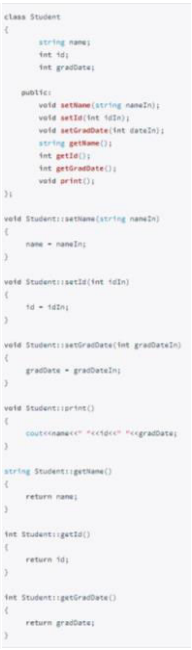
```cpp
class Student
{
        string name;
        int id;
        int gradDate;

    public:
        void setName(string nameIn);
        void setId(int idIn);
        void setGradDate(int dateIn);
};
```

So, we can set our data but now we need to 'get' the data. Traditionally these are called get-functions.

```
class Student
{
        string name;
        int id;
        int gradDate;

    public:
        void setName(string nameIn);
        void setId(int idIn);
        void setGradDate(int dateIn);
        string getName();
        int getId();
        int getGradDate();
        void print();
};
```

```
void Student::setName(string nameIn)
{
     name = nameIn;
}

void Student::setId(int idIn)
{
     id = idIn;
}

void Student::setGradDate(int gradDateIn)
{
     gradDate = gradDateIn;
}
```

```
string Student::getName()
{
     return name;
}

int Student::getId()
{
     return id;
}

int Student::getGradDate()
{
     return gradDate;
}
```



```
int main()
{
    int integer1;
    float float1;
    Student student1;

    integer1 = 4; //assign a value to integer1
    float1 = 4.333; //assign a value to float1

    student1.setName("Catherine Gamboa"); //assign a value to the student name
    student1.setId(54345); //assign a value to the student id number
    student1.setGradDate(2017); //assign a value to the student grad date

    //Let's print the values of our variables
    cout<<"integer1 = "<<integer1<<"\n";
    cout<<"float1 = "<<float1<<"\n\n";

    //There are two ways we can print the values of our class:
    //The first is to call the print function we created.
    cout<<"Using the Student::print function\n";
    cout<<"Student1 = ";
    student1.print();
    cout<<"\n\n";

    //The second is to access each member of the class using the get functions
    cout<<"Using the student access functions\n";
    cout<<"Student1 name = "<<student1.getName()<<"\n";
    cout<<"Student1 ID = "<<student1.getId()<<"\n";
    cout<<"Student1 Grad Date = "<<student1.getGradDate()<<"\n";


    return 0;
```

```
integer1 = 4
float1 = 4.333

Using the Student::print function
Student1 = Catherine Gamboa 54345 2017

Using the student access functions
Student1 name = Catherine Gamboa
Student1 ID = 54345
Student1 Grad Date = 2017
```

The syntax for a class is:

```
class ClassName
{
      member1;
      member2;
      ...

public:
     return variable accessFunction1(function parameters);
     return variable accessFunction2(function parameters);
      ...
} ;

returnVariable ClassName:: accessFunction1(function parameters)
{
    function statements;
}

returnVariable ClassName:: accessFunction2(function parameters)
{
    function statements;
}

...
```

C++ Convention is:

- Capitalize the first letter of the classname.

```
class ClassName
```

- Private members are listed first. If you do this, there is no need to use the 'private' keyword. If you list them after the public keyword, you will need to identify them using the private keyword.

```
{
      member1;
      member2;
      ...
```

- Use 'getVariableName' for accessing private variables Use 'setVariableName' for assigning values to private variables

```
public:
     return variable accessFunction1(function parameters);
     return variable accessFunction2(function parameters);
      ...
} ;
```

It is conventional to put classes in a header file. For this example we are going to put them in the main file so we can see both the main program and the class at the same time. Hopefully, this will make it a little easier to see what is going on in the program.

Notice we need to use the "dot operator" to access the public members.
For example, to get the value of the grad date in the student class we need to type:

```
studentVariableName.getGradDate();
```

There is another operator (->) that is used for accessing pointer members of a class. We will discuss

There is a special function member that we need to talk about, constructors.

A constructor is special function that is executed whenever we create a new instance of the class. It is used to set initial values of data members of the class.

For example, in our Cats class we may want to have an initial value for the age of a cat and it's breed. If we set initial values, we do not need to require the program or user set every value.

Constructors do not return a value, including void.

The declaration for a constructor is:

```
ClassName::ClassName();
```

The definition of a constructor is:

```
ClassName::ClassName()
{
     dataMemberName1 = value;
     dataMemberName2 = value;
     ...
}
```

In addition to constructors, C++ also has destructors.

Destructors are special class functions that are called whenever an object goes out of scope. Just like a constructor, a destructor is called automatically.

- Destructors **cannot**:

```
- return a value
- accept parameters
```

- Destructors must have the same name as the class.

The syntax for a destructor is similar to the constructor:

The destructor is identified with a tilda (~) symbol.

Declaring a destructor:

```
~className()  //this is a destructor
```

Defining a destructor:

```
classname::~classname()
{
     //tasks to be completed before going out of scope
}
```

One of the more important tasks of a destructor is releasing memory that was allocated by the class constructor and member functions.

C++ has a pointer called 'this'.

'this' returns its own address.

There are a few cases where 'this' might be necessary, but often using it is considered a stylistic preference.

Note in the program below, to compare the Area of the Shape's own area with the area of the shape

```
 //Use 'this' to compare areas
//The class functions
     int compareWithThis(Shape shape)
     {
        //return the area of the calling shape
        return this->Area() > shape.Area();
     }
```

```
  //Using the class function in a program
  //In this case sh1.Area() is being compared to sh2.Area()

  if(sh1.compareWithThis(sh2)) {
     cout << "\nShape2 is smaller than Shape1" <<endl;
  }
```

I can perform the exact same function and not use 'this'.

```
    //'this' is not necessary to compare shapes
     int compare(Shape shapeIn)
     {
        return Area() > shapeIn.Area();
     }
```

To use the class function:

```
if(sh1.compare(sh2))
{
   cout << "\nShape2 is smaller than Shape1" <<endl;
}
```

C++ allows class constructors to accept parameters. These parameters will set the values of class members when the object is created. Without using setFunctions.

```
    int a = 5;
    int b = 4;
    float f1 = 5.43;
    float f2 = 6.32;
    char c1 = 'c';
    char c2 = 'z';
    std::cout<<findSmallerInt(a,b)<<" is the smaller of "<<a<<" and "<<b<<"\n";
    std::cout<<findSmallerFloat(f1,f2)<<" is the smaller of "<<f1<<" and "<<f2<<"\n";
    std::cout<<findSmallerChar(c1,c2)<<" is the smaller of "<<c1<<" and "<<c2<<"\n";

    return 0;
}

int findSmallerInt(int input1, int input2)
{
    if(input1<input2)
        return input1;
    return input2;
}
float findSmallerFloat(float input1, float input2)
{
    if(input1<input2)
        return input1;
    return input2;
}

char findSmallerChar(char input1, char input2)
{
    if(input1<input2)
        return input1;
    return input2;
```

```
4 is the smaller of 5 and 4
5.43 is the smaller of 5.43 and 6.32
c is the smaller of c and z
```

RESET QUIZ | TEST RUN | SUBMIT ANSWER

Wow, that was a pain. The same code written three times with three different declarations and definitions!

> Function Overloading will allow us to use the same function name for different functions. As long as the argument list is different, the compiler will be able to choose the correct definition.
>
> Different argument list means either the variable type is different and/or the number of arguments is different.
>
> In the next example, I overloaded a function called findSmaller.
>
> This function will have three variations; integer arguments and return variable, float arguments and return variable, and character arguments and return variable.
>
> Now I can write a program using findSmaller with three different input variable types.

```
    int a = 5;
    int b = 4;
    float f1 = 5.43;
    float f2 = 6.32;
    char c1 = 'c';
    char c2 = 'z';
    std::cout<<findSmaller(a,b)<<" is the smaller of "<<a<<" and "<<b<<"\n";
    std::cout<<findSmaller(f1,f2)<<" is the smaller of "<<f1<<" and "<<f2<<"\n";
    std::cout<<findSmaller(c1,c2)<<" is the smaller of "<<c1<<" and "<<c2<<"\n";

    return 0;
}

int findSmaller(int input1, int input2)
{
    if(input1<input2)
        return input1;
    return input2;
}
float findSmaller(float input1, float input2)
{
    if(input1<input2)
        return input1;
    return input2;
}

char findSmaller(char input1, char input2)
{
    if(input1<input2)
        return input1;
    return input2;
```

main.cpp main.hpp

```cpp
/*Goal: look at a program, and see if we can make it more versatile*/

#include "main.hpp"


int main()
{
    Compare c;
    int a = 5;
    int b = 4;
    float f1 = 5.43;
    float f2 = 6.32;
    char c1 = 'c';
    char c2 = 'z';
    std::cout<<c.findSmaller(a,b)<<" is the smaller of "<<a<<" and "<<b<<"\n";
    std::cout<<c.findSmaller(f1,f2)<<" is the smaller of "<<f1<<" and "<<f2<<"\n";
    std::cout<<c.findSmaller(c1,c2)<<" is the smaller of "<<c1<<" and "<<c2<<"\n";

    return 0;
}


```

```
4 is the smaller of 5 and 4
5.43 is the smaller of 5.43 and 6.32
c is the smaller of c and z
```

We saw in the Classes lesson that we can have two kinds of constructors:

- those that do not have input parameters
- those that do have them.

There will be times when we would like to have both options in a class. Luckily, we can use Overloading to achieve it!

When create a class we can have two different constructors and the compiler will know which one we want.

The program below is an example of a class that has two kinds of constructors.

main.cpp main.hpp

```cpp
//header file for main.hpp

#include<iostream>
#include<string>
using namespace std;

class Square
{
    private:
        int length;
        int width;
    public:
        Square();
        Square(int lenIn, int widIn);
        int getLength();
        int getWidth();
};

Square::Square()
{
    length = 0;
    width = 0;
}

Square::Square(int lenIn, int widIn)
{
    length = lenIn;
    width = widIn;
}

int Square::getLength()
```

```
s1 dimensions are: 0,0
s2 dimensions are: 3,4
```

Overloading Operators

We can overload operators in C++. This is useful when using user defined objects.

In the example below we will use operators to overload the '+' sign.

Let's look at the example in detail.
You can see in the code below, the function must specify a return type. Then it must use the keyword 'operator' followed by the '+' sign.

```
//The function that overloads the '+' sign
int operator + (Shape shapeIn)
{
    return Area() + shapeIn.Area();
}
```

To use the overloaded '+' sign, we just have to use it with our user defined objects.

For example:

```
int total = sh1 + sh2;
```

In this case the '+' will add the areas of the two shapes.

```
    private:
      int length;     // Length of a box
      int width;

    public:
      // Constructor definition
      Shape(int l = 2, int w = 2)
      {
         length = l;
         width = w;
      }

      double Area()
      {
         return length * width;
      }

      int operator + (Shape shapeIn)
      {
          return Area() + shapeIn.Area();
      }
};

int main(void)
{
    Shape sh1;    // Declare shape1
    Shape sh2(2, 6);    // Declare shape2

    int total = sh1 + sh2;
    cout << "\nsh1.Area() = " << sh1.Area();
    cout << "\nsh2.Area() = " << sh2.Area();
    cout << "\nTotal = "<<total;
```

```
sh1.Area() = 4
sh2.Area() = 12
Total = 16
```

Templates Example

We saw how overloading can be used to create a function that is fairly generic. Unfortunately, we still have to write a function for each variable type. Wouldn't it be nice to just write a function once and use it over and over for different variable types?

Well, actually, we can do that.

C++ allows us to create generic functions using **templates**.

```
//Our generic function
template <typename T>  //tell the compiler we are using a template
T findSmaller(T input1,T  input2);

int main()
{
    int a = 54;
    int b = 89;
    float f1 = 7.8;
    float f2 = 9.1;
    char c1 = 'f';
    char c2 = 'h';
    string s1 = "Hello";
    string s2 = "Bots are fun";

    //Wow! We can use one function for different variable types
    cout<<"\nIntegers compared: "<<findSmaller(a,b);
    cout<<"\nFloats compared: "<<findSmaller(f1,f2);
    cout<<"\nChars compared: "<<findSmaller(c1,c2);
    cout<<"\nStrings compared: "<<findSmaller(s1,s2);
    return 0;
}

template <typename T>
T findSmaller(T  input1,T  input2)
{
    if(input1 < input2)
        return input1;
    else
        return input2;
}
```

The function declaration:

```
template <typename T>  //tell the compiler we are using a template

//T represents the variable type. Since we want it to be for any type, we
//use T
T  functionName (T parameter1,T parameter2, ...);
```

The function definition:

```
template <typename T>
T functionName (T  parameter1,T  parameter2,...)
{
    function statements;
}
```

```
template <typename T, typename U, typename V>
T functionName (U  parameter1, V  parameter2,...)
{
    function statements;
}
```

For example:

```
T getBigger(T input1, U input2)
{
    if(input1 > input2)
        return input1;
    return input2;
}
```

```
/*Goal: learn to use templates with multiple variable types.
*/

#include<iostream>
using namespace std;

template <typename T, typename U>
T getBigger(T input1, U input2);


int main()
{
    int a = 5;
    float b = 6.334;
    int bigger;
    cout<<"Between "<<a<<" and "<<b<<" "<<getBigger(a,b)<<" is bigger.\n";

    cout<<"Between "<<a<<" and "<<b<<" "<<getBigger(b,a)<<" is bigger.\n";
    return 0;
}

template <typename T, typename U>
T getBigger(T input1, U input2)
{
    if(input1 > input2)
        return input1;
    return input2;
}
```

```
Between 5 and 6.334 6 is bigger.
Between 5 and 6.334 6.334 is bigger.
```

Vectors and Iterators

Vectors are more versatile than arrays, and can be resized during runtime. We can also insert elements into a vector (we'll insert elements a little later in the lesson).

This characteristic adds a lot of freedom to vectors that we don't have with arrays. It also means we have to adjust how we access elements in a vector. Since we can add elements anywhere in the vector, we do not refer to the first element of a vector as the zero element, we call it the beginning. The last element is called end. To keep track of where we are in the vector, we need an **iterator**.

In the program below, you should notice the following:

- We use ::assign to add and define elements to the vector
- We instantiate an iterator for the vector class:

```
//creating an iterator for the vector
std::vector<int>::iterator it;
```

- We use the iterator to cycle through the vector. Begin is the first element in the vector, end it the last.

```
for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
  std::cout<<*it<<" ";
```

- Notice, we dereference the iterator to print out the value of the vector:

```
std::cout<<*it<<" ";
```

- Notice, we have to increment the iterator as we execute the for loop.

```
for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
```

vectorA.assign(n, m) → assigns n times, m value

insert elements

insert is another method to add elements to a vector.

insert adds elements to the location *after* the iterator.

To add elements using insert:

- Set the iterator to one before the location where you would like to add an element.
- Use insert to add the element.

The advantage of emplace is, it does in-place insertion and avoids an unnecessary copy of object. For primitive data types, it does not matter which one we use. But for objects, use of emplace() is preferred for efficiency reasons.

```
  //printing the contents of vectorInts
  std::cout<<"VectorInts has these elements:\n";
  for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
    std::cout<<*it<<" ";

  //insert an element after the first element
  it  = vectorInts.begin() + 1;
  vectorInts.insert(it, -1);
  std::cout<<"\n\nAfter the insert\n";
for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
    std::cout<<*it<<" ";

  //insert an element after the third element
  it  = vectorInts.begin();
  vectorInts.insert(it + 3, -2);
  std::cout<<"\n\nAfter the insert\n";
for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
    std::cout<<*it<<" ";

  //insert an element after the third element
  it  = vectorInts.begin();
  vectorInts.insert(it + 5, -3);
  std::cout<<"\n\nAfter the insert\n";
for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
    std::cout<<*it<<" ";

  return 0;
}


```

```
vectorInts has 8 elements
VectorInts has these elements:
0 1 2 3 4 5 6 7

After the insert
0 -1 1 2 3 4 5 6 7

After the insert
0 -1 1 -2 2 3 4 5 6 7

After the insert
0 -1 1 -2 2 -3 3 4 5 6 7
```

emplace Elements

Another method for adding elements is ::emplace.

Emplace puts an element in a vector at the position pointed to by the iterator.

Emplace differs from insert in the method used to insert the element. Insert **copies** the values of the vector while emplace does an **in-place insertion**. This means the insertion occurs at the point after the iterator.

The difference makes emplace more efficient than insert in special cases.

```
26    std::cout<<"VectorInts has these elements:\n";
27    for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
28      std::cout<<*it<<" ";
29
30    //insert an element after the first element
31    it  = vectorInts.begin() + 1;
32    vectorInts.emplace(it, -1);
33    std::cout<<"\n\nAfter the insert\n";
34  for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
35      std::cout<<*it<<" ";
36
37    //insert an element after the third element
38    it  = vectorInts.begin();
39    vectorInts.emplace(it + 3, -2);
40    std::cout<<"\n\nAfter the insert\n";
41  for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
42      std::cout<<*it<<" ";
43
44    //insert an element after the third element
45    it  = vectorInts.begin();
46    vectorInts.emplace(it + 5, -3);
47    std::cout<<"\n\nAfter the insert\n";
48  for (it = vectorInts.begin(); it != vectorInts.end(); ++it)
49      std::cout<<*it<<" ";
50
51    return 0;
52  }
53
54
```

```
vectorInts has 8 elements
VectorInts has these elements:
0 1 2 3 4 5 6 7

After the insert
0 -1 1 2 3 4 5 6 7

After the insert
0 -1 1 -2 2 3 4 5 6 7

After the insert
0 -1 1 -2 2 -3 3 4 5 6 7
```

QUESTION 1 OF 2

What will be the value of i and j after the following code is executed?

```
int i = 10;
int j = i++;
```

- [ ] i = 11, j = 11
- [x] i = 11, j = 10
- [ ] i = 10, j = 11
- [ ] i = 10, j = 10

Access Control for Inherited Classes

Let's look at this bit of code from our example:

```
//The derived class with Student as base class
class GradStudent : public Student
{
    private:
        string degree;
    public:
        GradStudent();
        void setDegree(string degreeIn);
        string getDegree();
};
```

Notice the line:

```
class GradStudent: public Student
```

The access control before the base class (in this case 'public') determines the access of the inherited class.

There are three types of access control: public, private, and protected.

- **Public Inheritance** means all public members of the base class are accessible to the derived class
- **Private Inheritance** means all members of the base class are private to the derived class
- **Protected Inheritance** means all members of the base class are protected to the derived class.

It is very rare to have a protected or private inheritance, the vast majority of the time inheritance is public.

In the program below, the inheritance is now private:

```
//The derived class with Student as base class
class GradStudent : private Student
{
    private:
        string degree;
    public:
        GradStudent();
        void setDegree(string degreeIn);
        string getDegree();
        void setStudentId(int idIn); //need this to access Student::setId()
        int getStudentId(); //need this to access Student::getId()
};
```

Now that we have a private inheritance, the Student member functions setId() and getID() are no longer available to the GradStudent class.

When we write the member functions, we must explicitly refer to the Student class.

```
int GradStudent::getStudentId()
{
    //We must access getId() as a private function
    return Student::getId();
}
void GradStudent::setStudentId(int idIn)
{
    //We must access setId() as a private function
    Student::setId(idIn);
}
```

Multiple Inheritance

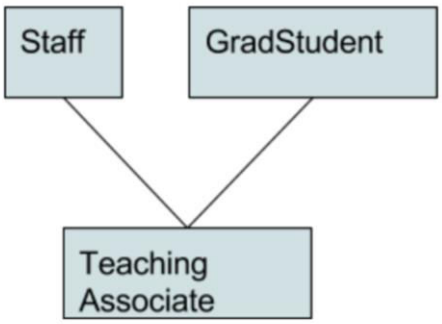
C++ classes can inherit from more than one class. This is known as "Multiple Inheritance".

The form for declaring multiple inheritance is:

```
class DerivedClass : access BaseClass1, ... ,access BaseClassN
```

For example, in the statement shown below, the derived class is TeachingAssociate. It inherits attributes from Staff and from GradStudent.

```
class TeachingAssociate: public Staff, public GradStudent
```



A graphical representation of multiple inheritance.