This course

- This is an intermediate CS class
- Basic building blocks of solving problems (C++)
- Gateway to other CS

Why C++

- Widely used
- Gateway PL

Algorithm vs Program

Complete this: "Computer science is about studying how to use _____ to solve problems."

- Algorithms: sequences are precise instructions that lead to a solution
- Programs: algorithms expressed in a language a computer understands

Computers

5 main components to computer: 1. Processor 2. Main memory 3. Inputs 4. Outputs 5. Secondary memory

Computer memory (2 parts) 1. Address: **where** 2. Data: **what**

How - Binary. - 8 bits = 1 byte. kB, MB, GB, TB

Basic C++

High-level computer languages

- Resembles "natural language"/algorithmic feel

Compilers

- language specific
- different compilers for the same language (gcc, clang)
- how do they work:
    1. source code
    2. object code

`[hello.cpp] --> g++ --> [hello]`

Intro to C++

Program style

C++ rules and conventions

Variable basics

- Variable: is a symbolic reference to data `int favNum = 0;`

```
 --------------  -----
Address         Value
 --------------  -----
favNum (0x100)   3
```

3 parts to every variable: 1. Type (int, char, strings, etc.) 2. Identifier 3. Value

Integers: - In C++, `int` takes up 4 bytes in memory. - -2,147,483,648 -- 2,147,483,647

- `unsigned int` 0 -- 4,294,967,295
- `short int` (2 bytes) -32,768 -- 32,767
- `unsigned short int` (2 bytes) 0 -- 65,535

Floating point: - `float` and `double` 122.5555550000000 122.55555556666

Boolean: - `bool` true or false - false --> 0, true --> non-0

Characters: - `char` , - `char letter = 'b';`

Strings: - `string question = "what is life?";` - `#include <string>`

In general, your variable types **must match** when operating over variables.

Variable naming: - A *good* variable name indicates what data stored in it - Noun, noun phrase. `firstName` - A *good* function name is verb, or verb phrase. `SortNumbers()` - Avoid `x` , or `var2`

- **Snake case** `crunch_numbers()`
- **Camel case** `CrunchNumbers()`

```
int t = 60;  // Time in minutes
int timeInMinutes = 60;
```

C++ Rules: (breaking results in a syntax error) - They **must** start with either a letter or `_` - The rest of the characters can be alphanumeric or `_` s - They cannot spaces or . or special symbols - They cannot be a **reserved keyword** - `int int;` `bool if;`

Variable comparisons:

```
a == b
a != b
a < b
a > b
a <= b
a >= b
```

The result is **always** a Boolean (true or false)

Variables and operators

Type compatibilities: - General rule: **you cannot operate on differently typed variables** - Exceptions: `int` and `double`

There are two rules for operating between `int` and `double` 1. Variables of type `double` should NOT be assigned to variables of `int` 2. Variables of type `int` can *normally* be stored in variables of type `double`

Boolean expressions:

- Note that `=` is not the same as `==`
  - `=` is for variable assignment
  - `==` is for variable comparison

Logical operators: - AND, `a && b` true if BOTH are true - OR, `a || b` true if EITHER are true - NOT, `!a` false if a is true (and vice versa)

```
AND
x y (x && y)
------------
F F    F
F T    F
T F    F
T T    T

OR
x y (x || y)
------------
F F    F
F T    T
T F    T
T T    T

NOT
x  !x
-----
F   T
T   F
```

Arithmetic expressions: - Precedence rules are same as algebra

`x + y * z` y and z are multiplied first `(x + y) * z` use parens to force order

Precedence for operations in C++: (highest to lowest) - unary `! ++ --` - binary arithmetic `* / %` - binary arithmetic `+ -` - boolean `< > >= <=` - boolean `== !=` - logical `&&` - logical `||`

Recommend using parentheses to remove ambiguity in operator order!

a + b (binary) !a (unary)

Increment/Decrement operators `++ --`

```
int foo = 0;

foo = foo + 1;
foo += 1;
foo++;
++foo;  // there's a small diff between ++foo and foo++

foo = foo - 1;
foo -= 1;
foo--;
--foo;
```

`crunchNumber(a,b)`

C++ Operator overloading! - `+` `-`

Command line fun

Directory structure:

```
              /(root)
    bin       usr       lib
     /\        /\
    a  b      c  d
```

Pathname: - Refers to the **location** of a file - **Absolute**: entire path name - **Relative**: the location *relative* to another location - Shortcuts: - `~` home dir - `.` current dir - `..` parent dir

Common commands: - `cd` change directory - `pwd` print working directory - `ls` list directory files - `cp` copy file (will overwrite) - `rm` remove file - `mv` move file (does not make a copy)

Text manipulation commands: - `cat` show contents of a file - `more/less` show contents one page at time - `head` show first 10 lines of a file - `tail` show last 10 lines of a file

Code Repositories:

- Sync our code between different computers
- GitHub is a service that lets us use git as a version control system

Why are we learning git in this class? - Collaborate with others on coding projects - Share code ownership with other students/TAs/instructors - Work on larger projects (more people) - Provide feedback/review code

Git concepts: - **repo** (short for repository): the place where your code (and its history) is stored - **Cloning** a repo: making a copy of a repo on another machine - **Commit** a change: when you make a change, you "solidify" the change by committing - **Push** the changes: after committing, we update our repot so others can see the changes - **Pull**: retrieve the most up-to-date repo

Typical workflow: 0. Clone the repo to your local machine `git clone` 1. Make changes to the files 2. Add changes `git add` 3. Commit changes `git commit` 4. Push changes to the remote repo `git push`

Control flow

The order in which statements get executed.

If/else (conditionals)

- **Branch**: How a program chooses between 2 alternatives (2 blocks of statements)

```
if (Boolean expression)
{
    // statement 1
}
else
{
    // statement 2
}
```

Again, beware '=' or '=='

Loop basics (while, for, do-while)

```
int countDown = 3;
while(countDown > 0)
{
    cout << "Alright ";
    countDown--;
}
```

> Alright Alright Alright

```
int flag = 1;
do
{
    cout << "Yo";
    flag -= 1;
}
while (flag>0); // note the semicolon
```

> Yo

```
/*
    For loop
    Similar to while loop, but with extra parameters

    for (declare counter variable; loop condition; modify counter variable)
    {
        loop body;
    }
*/
for (int count = 2; count<6; count++)
{
    cout << "Apples ";
}
```

> Apples Apples Apples

```
//Cool trick with while loops and ++
int max = 0;
while (max<4)
{
    cout << "1+";
    max++;
}

int max=0; //an equivalent way of doing this
while (max++<4)
{
    cout<<"1+";
}
```

> 1+1+1+

1. start the while loop condition check
2. grab the value in max (check max<4)
3. THEN increment max

++max: 1. increment the value in max 2. then USE it

```
int foo = 1;
while (foo>0)
{
    cout << foo << endl;
}
```

> 1 1 1 ...

Infinite loops: Avoid: the loop body should contain some statement that will eventually cause the loop condition to turn false.

Common use for 'for' loops: Sums, accumulating results

```
int sum = 0; // MUST INITIALIZE IT BEFORE THE LOOP
for (int count=0; count<5; count++)
{
    int num;
    cin >> num;
    sum += num;
}
cout << "sum is " << sum << endl;
```

Loop advice: - A 'for' loop is generally a good choice when there is a predetermined number of iterations - For cases when you don't, use a 'while' loop

3 common ways to end a while loop: 1. End with "sentinel value - Use a particular value to signal when the loop is done 2. Ask before iterating - Ask if the user wants to continue 3. Running out of input - Use `eof` function to indicate the end

```
// sentinel value
int number;

cout << "enter a non-negative integer and I'll double it!\n"
     << "enter a negative integer to quit";
cin >> number;
while (number >= 0) // number is the "sentinel"
{
    cout << "Doubled! " << number*2 << endl;
    cout << "enter a non-negative integer and I'll double it!\n"
         << "enter a negative integer to quit";
    cin >> number;
}
```

```
// Ask before iterating
char answer;
cout << "Are you happy? (y/n) ";
cin >> answer;

while ((answer == 'n') || (answer == 'N'))
{
    cout << "How about now? ";
    cin >> answer;
}
```

Nested loops: - The body of a loop can contain any kind of C++ statement - When the loops are nested, **all iterations of the inner loop execute for each iteration of the outer loop**.

Example: - we want to collect students' grades - students have multiple assignments (quiz, hw) - we want to output class average - we'll go through each student, one at a time, and calculate their scores

```
 int students(10);
double grade(0), subtotal(0), grandTotal(0);

for (int count=0; count < students; count++)
{
    cout << "student number: " << count << endl
         << "Enter grades (to finish, enter a negative number)\n";
    cin >> grade;
    while (grade >= 0)
    {
        subtotal += grade;
        cin >> grade;
    }
    cout << "total grade for student " << count << " is "
         << subtotal << endl;
    grandTotal += subtotal;
    subtotal = 0;
}

cout << "class average " << grandTotal/students << endl;
```

Nested conditionals

Switch statements

Intro to functions in C++

- Recall: a *block* of code is a series of statements that is enclosed by `{}`
- Variables *declared* in that block of code are *local* to that block
    - **scope**

Programmer-defined functions

1. **Input arguements**
2. **Output** (or not)

- You have to declare the *return type* of the function
    - The type of data is returns
    - We also have to declare the types of the *arguments* of the functions
    - `double crunchNumbers(int x, int y);`
    - `string crunchNumbers(int x, int y);`
    - `void crunchNumbers(int x, int y);`

2 components: 1. Function declaration - Just like variables, a function must be declared before it's used - **Must** be placed outside `main()` (usually before it) - **Must** be placed *before* the function is **defined** and **called** 2. Function definition - Where you implement the function (the details) - **Must** be placed outside of `main()` - Can be before or after `main()`

Void functions: - Sometimes we want to design subtasks to be implemented as functions - Maybe there is some repetition (like printing debug/log statements) - We may not want to return anything - We can use `return` to signal an "interupt" and end the function early

Function declaration

Function definition

Pre-defined functions in C++

- C++ comes with lots of "built-in" libraries of pre-defined functions.

Function calls

- When we call a function, our arguments are getting pass as **values** into the function.
    - `fun(a,b)` ; this passes the **values** of a and b into fun
    - in `fun`, `a` and `b` are treated as local variables to `fun`

Call-by-value vs Call-by-reference

- Usual way is to pass variables into function is "call-by-value"
- We can *also* call a function with arguments used as **reference** to the actual variable in *memory*
    - So we're not passing the variable itself, but its *memory location*
    - **Why do we want to do this?** Variables inside functions are local to those functions. What if we wanted them to change outside of it?
- Call-by-reference parameters:
    - allows us to change the *variable* used in the function call
- Call-by-value parameters do *not* change the variable used in the function call

Command-line arguments in C++

To use command-line arguements in our programs we add **2 arguments** to our `main()` - `argc` and `argv`

```
int main(int argc, char* argv[]){


}
```

- `argc` is the number of arguments
- `argv` is the full list of argument values as an array

Arrays

- An array is used to process a collection of data of the *same* type
    - Examples: list of names, list of measurements
- Why do we need arrays?
- Arrays are stored in memory (just like other variables)
- When reserving memory space for an array, the compliler needs to know 3 things:
    1. A starting memory address (location)
    2. How many elements
    3. What data type of the array elements

```
 char a[4];
// let's say a character takes 2 bytes of memory

address data
-------|-----
1023   | a[0]
1024   |
1025   | a[1]
1026   |
1027   | a[2]
1028   |
1029   | a[3]
1030   |

To look up a[3]: 2 x 3 = 6, 6 + 1023 = 1029.
To look up a[4]: 2 x 4 = 8, 8 + 1023 = 1031.
```

- What happens if we try to access `a[4]`? Array index out of range (error)
- Using an out of range index value **does not** always produce an error!
    - It can produce a warning, but may fail during runtime
- `int a[6]` valid range: 0...5

Default values - `int foo[10] = {2, 31};` - Extended initializer list (C++11 or later) - Fill in remaining values with "zero" values for that data type

Range-based `for` loops

Arrays in functions - Indexed variables can be arguments to functions - passing an array into a function - the array parameter is indicated by [] - it's usually necessary to pass the array AND its size - we can think of a[] as a variable that contains the "memory address" of the start of the array (location of a[0]) - an array parameter behaves similarly to a call-by-reference parameter

Constant array parameters - Array parameters allow a function to change the values stores in the array (similar to call-by-reference parameters) - If we want a function to **not change** the values of the array argument, we use the `const` keyword - A "constant array parameter"

Multidimensional arrays: - C++ allows arrays with multiple index dimensions (same type) - Example: `char page[30][100];` - page has 2 index values - the first 0--29 - the seconds ranges from 0--99 - `page[2][49]` - visually this looks like a matrix with 30 rows and 100 columns - `page` is an array of size 30, but page's base type is an array 100 characters

```
[0][0] [0][1] ... [0][98] [0][99]
[1][0] [1][1] ... [1][98] [1][99]
...    ...    ... ...    ...
[29][0] [29][1] ... [29][98] [29][99]
```

Multidimensional array parameters in functions: - Recall that the size of a one dimensinal array is not needed in the function declaration - `void displayLine(char a[], int size);` - But, the *base type* must be specified for a multidimensional array in a function parameter - `void displayPage(char page[][100], int size_rows);`

Compilation

- `g++` is composed of a number of smaller programs
- `ld` (linkage editor) merges one or more object files
  - Code written by others or using libraries (these are provided as object files)

```
[mda.cpp] --> (g++) --> mda
               |
               V
[mda.cpp] --> [mda.s] --> [mda.o] --> mda (executable)
             assembly    object
```

Make

- Is a *build automation* tool
  - Automatically builds executable programs and libraries from source code
  - The instructions live in a file called `Makefile` / `makefile`
- Put all the instructions we need to build the program in `makefile`
  - There is a syntax
  - Convention is that there's *one* `makefile` per project

```
# ex1.cpp ex2.cpp
# This is a comment

all: Exercise1 Exercise2

# This compilation command forces the compiler to use the C++ 11 standard.
# -Wall flag shows all warnings (not just errros)
Exercise1: ex1.cpp
    g++ ex1.cpp -o ex1 -std=c++11 -Wall

Exercise2: ex2.cpp
    g++ ex2.cpp -o ex2 -std=c++11 -Wall

clean:
    rm *.0 ex1 ex2
```

```
$ make #run the "all" command
$ make Exercise1
$ make clean
```

- Highly recommend getting in the habit of using make
- Get familiar with the syntax (our labs will have instructions on how to use make).

C++ programming with multiple files

- Real world C++ programming separates these pieces out (functions declarations/definitions/main/etc)
  - Using **header** files `.h` -- these contain function prototypes (declaration)
  - One or more files with function *definitions*, some with `main` and some might not

Why? 1. Reusability: - Using generic functions to avoid rewriting code 2. Modularization - Create stand_alone pieces of code - These can contain sets of functions or classes (or both) - A library is a module that is already compiled (it is object code) 3. Independent work flows - If we have multiple people working on a project, helpful to break it down into pieces to prevent collisions 4. Faster re-compilation and debugging - When you make a change, you only have to re-compile the parts that changed - Easier to debug a portion of the whole program

Lab 4 is our first example of this: - `constants.h` --> global variable declarations - `headers.h` --> function prototypes - `arrays.cpp` --> main, all function definitions - `ArrayFile.txt` --> external data file

Debugging

Debugging loops - Common errors with loops: 1. Off-by-one error: loop executes one-too-few or one-too-many times - Fix: Check your comparisons. Should it `<` and `<=` ? - Fix: Check your variable initializations. 2. Infinite loops: usually from a mistake in the boolean expression that controls the loop - Fix: Check the direction of inequalities `>` or `<` . Sometimes an issue with loops that use `!=` or `==` (with numeric values) - `(i <= 0)`

Loop tracing - **Trace** variables in a loop to observe *how* they change - Just insert `cout` statements in our loop body with the variable we care about.

Loop testing guidelines - Every time a program is changed, it should be re-tested - Changing one part may require changes to another part - Every loop should at least be tested using input to cause: - Zero iterations of the loop body - One iterations of the loop body - One less than the max number of iterations - Max number of iterations - If all of these are OK, then you have a robust loop

Starting over - Sometimes it is more efficient to throw out a buggy program and start over! - The new program will be easier to read - Likely be less buggy - You may develop a working program faster - The lessons learned in the buggy code will help you write a new better program

Testing and debugging functions - Each functions we write should be tested as a separate unit - "Driver" or "test" programs to run tests on our functions

Stubs - When a function being tested *calls* another functions that's not yet implemented/tested, use a *stub*! - A **stub** is a simplified version of a function - Stubs should be so simple that you have full confidence it will perform correctly

Debugging advice: - Keep an open mind - **Don't randomly change code** - Try explaining the program to someone else.

General debugging techniques: - Check for common errors, e.g. - Local vs reference parameters - `=` or `==` - `&&` instead of `||` - Typically are logic errors - Localize the error - Using tracing and stubbing - The textbook has great debugging examples

Pre- and Post-Conditions - Pre-conditions: what must be true before we call a function - Post-conditions: what the function will do once it is called. - the effect of function - changes to call-by-reference parameters

Test-driven development (TDD)

String

- The string class allows us to treat strings like a basic data type.
  - Found in library
  - The "legacy" alternative is to treat strings as arrays of characters (from C language), `char message[6]`, c-strings.
- Comes with useful functions like `length()`, `substr()`, `find()`, `replace()`, etc.
- Since strings are made of characters, we can index individual characters
- Use the + operator to concatenate strings
- Use the += operator to append to a string (i.e. add to the end)
- Conversions

1. String to number: stoi() for integer; stod() for floating point
2. Numbers to strings: to_string()

- Character functions: operate on individual characters
  - Found in library
  - Character manipulators: tolower(), toupper() -- returns ascii value (integer) of lower case character
    - Note: these operate on single characters!
    - How can we use this on an entire string?
  - Character inspectors:
    - isspace() -- boolean function returns true if the argument is whitespace (space, tab, newline)
    - isupper(), islower() -- checks if upper/lowercase letter
    - isdigit() -- checks if 0–9
- Search functions
  - find() will search for the first occurance of a string inside another string -- returns first index

```
string question = "How now brown cow?";

int first = question.find("ow");
```

  - find() has an optional parameter of the starting index: find(string, index)
  - finds the first occurance starting at position `4`

```
int next = question.find("ow", 4);
```

  - You can use find to check if a substring is not in the target string
  - string::npos is the "no position"; is returned if no position exists

```
if (question.find("banana") == string::npos)
{
    cout << "Yes, we have no banana\n";
}
```

    - rfind() will search the last occurance of a string
- length() returns the length of the string
    - useful for looping over a string
    - `string.length()`
- String manipulators:
    - replace(start_position, number_of_places_to_replace, replacement_string)
    - insert(start_position, insertion_string)
    - substr(start_position, number_of_places_after_start)

Searching and Sorting

- Very common algorithms in CS

Sequential search

- AKA 'linear search', start from the beginning of our array/list and scan until we find our target

Given: array of integers, integer target 1. Go through each element of the array, starting from the first 2. For each element: if it is equal to the target, then return the index of the array 3. If the target is never found, return -1

- Pros and cons of sequential search Pros: simple, easy Cons: only finds the 1st occurance of the target; slow
- There's a faster search algorithm out there (we'll get into this later)

Sorting

- Sorting a list of values is very common task
    - Create an alphabetical listing
    - In general, sorting data in ascending/descending order
- Many sorting algorithms
    - Some are efficient, some are easier to understand

Selection sort

- Idea: when the sort is complete, the elements of the array are ordered in ascending order:
    - `a[0] <= a[1] <= a[2] ... <= a[size-1]`
- The outline of the algorithm:

```
for (int i=0;i<size;i++)
{
    place the ith smallest element in a[i]
}
```

1. Searth the smallest value in the array
2. Place this value in `a[0]` and place the value that was in `a[0]` into the location where the smallest value was found (swap)
3. Start from `a[1]`, find the smallest remaining value, swap it with `a[1]`
4. Starting from `a[2]`, continue, ..., until the array is sorted.

```
a[0] a[1] a[2] a[3] a[4] //swap a[3] with a[0]
 8    6    10   2    16

a[0] a[1] a[2] a[3] a[4] //from a[1], it's smallest, nothing to swap
 2    6    10   8    16

a[0] a[1] a[2] a[3] a[4] //swap a[2] with a[3]
 2    6    10   8    16

a[0] a[1] a[2] a[3] a[4] //sorted
 2    6    8   10    16
```

```
int indexOfSmallest(int a[], int startIndex, int size)
{
    int min = a[startIndex];
    int indexOfMin = startIndex;

    for (int index=startIndex+1;index<size;index++)
    {
        if (a[index]<min)
        {
            min = a[index];
            indexOfMin = index;
        }
    }
    return indexOfMin;
}

void swapValues(int &v1, int &v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}

void selectionSort(int a[], int size)
{
    int nextSmallest;

    for (int index=0;index<size-1;index++)
    {
        nextSmallest = indexOfSmallest(a, index, size);
        //swap index with nextSmallest
        swapValues(a[index], a[nextSmallest]);
    }
}
```

Bubble sort

- Similar to selection sort in terms of efficiency
- It keeps exchanging (sorting) 2 adjecent elements at a time
    - When no more exchanges are required; the array is sorted

```
(5 1) 4 8
(1 5) 4 8
1 (5 4) 8
1 (4 5) 8
1 4 (5 8)
(1 4) 5 8
1 (4 5) 8
1 4 (5 8)
```

```
void bubbleSort(int a[], int size)
{
    for (int i=size-1;i>=0;i--)
    {
        for (int j=1;j<=i;j++)
        {
            if (a[j-1]>a[j])
            {
                swapValues(a[j-1],a[j]);
            }
        }
    }
}
```

C-String sidebar

- A C-string is declared as arrays of `char`
- But, an array of `char` is not by itself a C-string. A valid C-string requires a terminating "null" character.
    - The null character has an ASCII value of 0
        - The escape sequence `\0` or `\x00`

```
char foo[10]; //Character array - can hold a C-string, but it is not yet a valid C-string (no terminating character)

char bar[10] = {'h', 'e', 'l', 'l', 'o', '\0'}; //array initialization

char baz[10] = "hello"; //shortcut for initialization

/*
In memory
0  1  2  3  4  5  6  7  8  9
h  e  l  l  o  \0
*/

char shaz[10] = ""; //Empty C-string of length 0 equal to ""

/*
In memory
0  1  2  3  4  5  6  7  8  9
\0
*/
```

- Since `char` is a built-in data type, no header file is required to make C-strings.
    - The C library `<cstring>` contains a number of useful functions that operate on C-strings `strncpy()`, `strcmp()`.

Binary search

- Binary search assumes the input data is already sorted.
    - `a[0] <= a[1] <= ... <= a[size-1]`
- Twist: since the array is sorted, let's split it in half! How does the target value compare with the median value?

```
Target 13

Indices: 0  1  2  3  4  5  6
Values:  2  3  5  7  11 13 17

13 > 7: take second half of the list
Indices: 4  5  6
Values:  11 13 17

13 <= 13: found it! 2 comparisons instead of 6
```

```
//Pre-condition: array is sorted in ascending order
//Post-condition: if target is found, return non-negatice index; otherwise return -1
/*
    High-level idea:
    1. Start by looking at the item in the middle of the list
    2. If the number = our target, done!
    3. If it is greater than our target, look in the 1st half of the list
    4. If it is less than our target, look in the 2nd half
    5. repeat
*/
int binarySearch(int a[], int size, int target)
{
    int first = 0;
    int last = size-1;
    int mid;

    while (first<last)
    {
        mid = (first+last)/2;
        if (target==a[mid])
        {
            return mid;
        }
        else if (target<a[mid])
        {
            last = mid-1;
        }
        else
        {
            first = mid+1;
        }
    }

    return -1;
}
```

File I/O

- Input and output via files

```
[input file] --> [C++ program] --> [output file]
```

How to do this? - *Stream* variables for file I/O - Must be declared before we can use file I/O - Must be initialized before we can work with the files - Initialization of a stream means connecting it to a file - Can have their values changed - E.g. disconnecting a stream from one file and then connecting it to another file

- Declare an input-file stream variable `ifstream inStream;`
- Declare an output-file stream variable `ofstream outStream;`
- Connect to a file
    - Means "opening" the file
- Use the member function called .open() `inStream.open("infile.dat");` `outStream.open("outfile.dat");`

Errors on opening files - It might not exist - Incorrect file path - Wrong permissions

.fail() --- tests the success of a stream operation - returns true if the stream operation filed

- Using the input stream
- Once connected to a file, get input from the file using the extraction operator `>>` (just like using cin)
- Once connected to a file, write output to the file using the insertion operator `<<` (just like using cout)

Close the file! - Closing disconnects the stream from the file - `inStream.close()`, `outStream.close()` - Reduces the chance of a file being corrupted if the program terminates abnormally - The system will automatically close the files you forget, as long as your program ends normally (with runtime errors)

Detecting the end of a file: - Files may vary in length - Programs may not be able to correctly assume the number of lines/items in the file 1. Using the >> operator 2. member function .eof()

Method 1

```
 double next, sum(0);
int count = 0;

while (inStream >> next)
{
    //The loop condition does 2 things:
    //1. Reads a value from the inStream object and stores it in the variable next
    //2. Then, returns a boolean value
    //True if the value can be read and stores
    //False if there is not a value to be read (i.e. end of file)
    sum += next;
    count++;
}
cout << "Average: " << sum/count << endl;
```

Method 2 .eof()

```
 char c;
inStream.get(c);
while(!inStream.eof())
{
    cout << c << "-";
    inStream.get(c);
}
cout << "\nDone!\n";
```

Which should we use? - Use `.eof()` when input is treated as text/strings/characters while using .get() to read the input - Use the extraction operator (>>) when the input is numerical data

Member function .get() - This is a function of every stream variable (for cin or any stream object) - Reads one character at a time - Stores that character in a variable of type `char` (passed in as argument) - Does not use the >> operator - Does not skip whitespaces

Difference between .get() and getline() - getline() lets us input streams with whitespaces - Unlike cin, which separates by whitespaces - getline() is not a member function. It's provided by the `<string>` library. It takes as argument the input stream variable

Appending data - If the output file that you want to write already contains data, we would be overwriting it!

- To append new output to the end of an existing file, use the constant `ios::app` defined in the library.
    - We do this when we open the file
    - If the file does not exist a new file is created
    - `ios::app` is just another C++ constant that means "seek to the end before each write"

Formatting output to file - Recall format decimal precision for standard out

```
 cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(3);
```

- We can do the same thing for our output streams

```
 outStream.setf(ios::fixed);
outStream.setf(ios::showpoint);
outStream.precision(3);
```

File stream objects in functions - We can define functions that do I/O - But we have to pass the stream variables by reference into the function

HW3 sidebar

- Difference between `++i` and `i++`
- `++i` will increment `i` and return incremented value
- `i++` will increment `i` and return the original value (before it was incremented)

```
int i = 1;
int j = ++i;
// i == 2, j == 2

i = 1;
j = i++;
// i == 2, j == 1
```

Structs and classes in C++

- A class ia a complex data tyle that can have:
    - Multiple values within it, and multiple member functions
    - E.g. `string` class has member functions (`.length(), .erase(), .find()`)
- Structures are a good intro to classes
    - Use them as "containers" (objects) for variables of (possibly) different types
    - We call these member variables
    - Structs do not have member functions (those come with classes)
- Idea is it is a contianer that holds multiple values and these are logically related to one another, and should be packaged as a single item
    - E.g. A bank investment account comes with:
        - a balance
        - an intersest rate
        - a term (how many months)
    - E.g. A student record
        - ID number
        - last name
        - first name
        - GPA

```
struct Account
{
    double balance;
    double interest_rate;
    int term;
    double getData();
};

double Account::getData() { return balance;}
```

Using 'struct's - Definitions are usually placed outside any function definitions - So they can be globally used

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

struct Account
{
    double balance;
    double interest_rate;
    int term;
};

void getData(Account &account)
{
    cout << "Enter account balance: ";
    cin >> account.balance;
    cout << "Enter account interest rate: ";
    cin >> account.interest_rate;
    cout << "Enter the term (in months): ";
    cin >> account.term;
}

//Structs as return types
Account makeAccount(double theBalance, double theRate, int theTerm)
{
    Account temp;
    temp.balance = theBalance;
    temp.interest_rate = theRate;
    temp.term = theTerm;
    return temp;
}

int main()
{
    //To declare out objects of our struct
    Account myAccount, yourAccount;
    //Each object has distinct member variables for balance, term, etc.

    //Specifying member variables using "dot" operator
    getData(myAccount);

    double rate_fraction, interest;
    rate_fraction = myAccount.interest_rate / 100.0;
    interest = myAccout.balance * rate_fraction * (myAccount.term / 12.0);
    myAccount.balance = myAccount.balance + interset;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "In " << myAccount.term << "months, your account will have a balance of $" << myAccount.balance << endl;

    yourAccount = makeAccount(10000, 1.6, 6);
    cout << "Your account balance: " << yourAccount.balance << endl;

    return 0;
}
```

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

struct Date
{
    int month;
    int day;
    int year;
};

//Hierarchical structs
//Structs can contain structs as member variables
struct Person
{
    double height;
    double weight;
    Date birthday;
    string fullName;
};

struct House
{
    string city = "nowhere";
    string street;
    int zipcode;
    bool hasFirePlace;
};
House MuhHouse = {"SB", "State St", 93101, true}; //is this valid?
// Yes. It is the same as when we do: 'int number = 42;'

int main()
{
    //Has to match the order of the struct's definition
    Date dueDate = {2, 16, 2021};
    //Data birthday = {12, 24, 2000, 11}; //will this work? No
    Date birthday2 = {1};
    //will this work? Yes.
    cout << "birthday " << birthday.month << "," << birthday.day << "," << birthday.year << endl;

    Person person1 = {60, 90, {12, 1, 2000}, "Toddy"};
    cout << person1.fullName << ": " << person1.birthday.year << endl;

    cout << "My house street " << MuhHouse.street << endl;
    MuhHouse.street = "Main st";
    cout << "My house street now " << MuhHouse.street << endl;
    House friendHouse;
    cout << "Friend house city is " << friendHouse.city << endl;

    return 0;
}
```

Classes

- A `class` is a complex data type whose variables are objects
- The definition includes:
    - Member variables
    - Member functions

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

class DayOfYear
{
    public:
        int month;
        int day;
        void output();
};
/* Member functions are declared in the class definition */

/*
    Member function definition - identifies the class in which the function is a member of.

    The '::' operator ("scope resolution operator")
*/
void DayOfYear::output()
{
    cout << "month = " << month << ", day = " << day << endl;
}

int main()
{
    DayOfYear today, birthday;
    cout << "Enter month: ";
    cin >> today.month;
    cout << "Enter day: ";
    cin >> today.day;

    cout << "Today's day is ";
    today.output();

    return 0;
}
```

Limitations with `DayOfYear` - Changing how the month is stored requires changes to the while program - Example: if we decide to store the month as a 3 character value "JAN", "FEB", etc. - `cin >> today.month;` will no longer work - Any comparisons that assume the member variable is int will have to change - The `output` function may have to change

Ideal class definition - Changing the implementation details of `DayOfYear` requires changes to the program that uses objects of `DayOfYear` - Not ideal - An ideal class definition of 'DayOfYear' could be changed without requiring changes to the program that uses it

Fixing `DayOfYear` - We can do 2 things: 1. Add member functions to use when changing or accessing the member variables - If the program never directly references, then changing how the variables are stored will not require changing the program! 2. Be sure that the program does not directly reference memeber variables

Public or Private? - C++ helps us restrict the program from directly referencing member variables - By default, all members of a class are private - But we can make the public - Private member variables of a class can only be referenced within the definitions of member functions of that class, not by the progran! - Private members can be variables or functions

```
class DayOfYear{
    public:
        //void input();
        void output();
        void set(int newMonth, int newDay);
        //int getMonth();
        //int getDay();
    private:
        int month;
        int day;
};

void DayOfYear::set(int newMonth, int newDay)
{
    month = newMonth;
    day = newDay;
}

int main()
{
    DayOfYear today, dueDate, tomorrow;
    int m, d;

    cout << "Enter month: ";
    cin >> m;
    cout << "Enter day: ";
    cin >> d;
    today.set(m, d);

    cout << "Today's day is ";
    today.output();

    //Assignment operator between class objects
    tomorrow.set(2, 19);
    dueDate = tomorrow;
    dueDate.output();

    return 0;
}
```

Using private variables: - It is convention to make all member variables private - Private variables require member functions to perform all changing/retrieving of values - Accessor functions: obtain values of member variables and return them - Mutator functions: changes the values of member variables (`set()`)

Constructors

- A constructor is used to initialize member variables when an object is declared
- A constructor is a member function that is usually public
- A constructor is automatically called when an object is declared
- A constructor's name must be the same name of the class
- A constructor cannot return a value
    - No return type, not even void, it's a special class-specific function

```
#include <iosteam>
#include <string>
using namespace std;

class Cat
{
    private:
        int age;
        string name;
    public:
        int getAge();
        string getName();
        Cat(int a, string n);
        Cat(string n);
        Cat(int a);
```

```
        Cat(int a);
        Cat();
};

//Defining a constructor
Cat::Cat(int a, string n)
{
    age = a;
    name = n;
}

/*
    Overloading constructors.
    - You can have multiple constructors for the same class
    - Constructors can be overloaded by defining constructors with different parameter lists
*/

Cat::Cat(string n)
{
    name = n;
    age = 0;
}

Cat::Cat(int a)
{
    age = a;
    name = "cat";
}

//Default constructor
Cat::Cat()
{
    age = 0;
    name = "cat";
}

/*
    Member initialization.
    - C++11 (and beyond)
    - Simply set member variables in the class to some defauly value
*/
class Coordinate
{
    private:
        int x = 0;
        int y = 0;
    // ...
};

int main()
{
    Cat kitten(0, "grizabell");
    //We do NOT call constructors like this:
    //Cat kitten2.Cat(1, "snaggles");
    Cat kitten1, kitten2(13), kitten3("gregorson");

    int i = 0, j(0);

    Coordinate point; //already initialized x and y to 0

    return 0;
}
```

Initialization sections: - An alternative way to initialize member variables

```
Coordinate::Coordinate() : x(0), y(0) { }
```

Abstract data types (ADT)

- A data type consists of one or more values together with a set of basic operations defined on those values
    - `int`
    - You know how it is used, but we don't need to know how the computer deals with it internally (the implementation details)
- A data type is an abstract data type if programmers using it do not have to access the details of how the values and operations are implemented.
    - I.e. they are "abstract" to the programmer
- Separate the specification of how the type is used from the implementation
    - So the programmer (user) doesn't see the "insides" and doesn't need to
- This means:
    - Make all member variables private
    - Basic operations a programmer needs should be public member functions

ADT benefits - Changing an ADT implementation does not require changing a program that uses the ADT - ADTs make it standard to divide work among different programmers - One can write the ADT - One can write code that uses it - Standards and conventions in programming come up all the time in software development

Inheritance

- Inheritance refers to derived classes
    - Derived classes are classes that are obtained from another class
    - A derived class inherits the member functions and variables from its parent class (without rewriting them)
- Example:
    - `cin` belongs to the class of "all input streams" but not the class of input-file streams
    - I/O file streams (`ifstream` , `ofstream`) are actually derived classes from a general stream class, but with added features (member functions) like `open()` and `close()`

Example: - Natural hierarchy of bank accounts - Most general type of bank account: "Bank account": it stores a balance - A checking account is a Bank account that allows writing checks - A savings account is a Bank account but it has no checks but a higher interest rate - The more specific types of accounts retain the same foundation as the general account, but adds some features

Inheritance relationships: - The more specific class is called a derived or child class - The more general class is called the base, super, or parent class - If class B is derived from class A, then: - Class B is a derived class of class A - Class B is a child of class A - Class A is the parent of class B - Class B inherits the member functions and variables of class A

Derive: public or private? - Public: preserves all access levels for the inherited member functions and variables - Private: makes all inherited member functions and variables from the parent to be private

Protected members are accessible in the class that defines them and in classes that inherit from that class

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

//this will be our parent class
class Vehicle {
    private:
        string name;
        string color;
        int numberOfWheels;
        bool isNameEmpty() { return name == "";}
        bool isColorEmpty() { return color == "";}
        bool checkWheels() { return numberOfWheels>0;}
    public:
        Vehicle() : name(""), color(""), numberOfWheels(0) {}
        void setName(string n) {
            if (!isNameEmpty())
                name = n;
        }
        void setColor(string c) { color = c;}
        void setWheels(int wheels) { numberOfWheels = wheels;}
        void print();
};

void Vehicle::print() {
    cout << "Name: " << name << endl;
    cout << "Color: " << color << endl;
    cout << "Number of wheels: " << numberOfWheels << endl;
}
```

```
//this will be our child class of vehicle
//This will inherit all the member functions and variables from the Vehicle class
class Bicycle : public Vehicle {
    public:
        Bicycle(bool e) {
            setWheels(2);
            isElectric = e;
        }
        void setElectric(bool e)
        {
            isElectric = e;
        }
    private:
        bool isElectric;
};

int main()
{
    //declaring class objects
    Vehicle semi;
    Bicycle myBike(true);

    //set member variables using public member functions
    semi.setName("Big rig");
    semi.setColor("Red");
    semi.setWheels(18);

    //Same thing. but ...
    //setName and setColor are Vehicle class member functions pass to Bicycle via inheritance!
    //setElectric is only a Bicycle member function and can only be used by Bicycle objects
    myBike.setName("Bichael");
    myBike.setColor("Blue");
    myBike.setElectric(false);

    Vehicle vs[] = {semi, myBike};
    for (int i=0;i<2;i++)
    {
        vs[i].print();
    }

    return 0;
}
```

Test-driven development (TDD)

- TDD is a software development process (popular in industry)
- Relies on the reptition of very short development cycles:
    - Come with requirements (not code) first
    - Turn requirements into very specific test cases (still not code)
    - Now write code and test it with the test cases
    - Improve code until the tests pass

TDD "lite" - Write test code and "actual" code side-by-side so your implementation is always tested - Write the simplest test case and make your code pass that case - Write another test case, expect your code to fail, see it fail, then add code to pass that test (and the previous one, ...) - With every new test case, we have to make sure that all our previous tests still pass.

Example

Requirement: - Write a function that "draws" (in ASCII characters) a square using `*` characters, given some integer input for the size. - Example `drawSquare(5)` draws:

```
 *****
*****
*****
*****
*****
```

First step: Write a test case for this requirement 1. Something to check on the expected value vs the actual 2. Something to run this check on a test of the function

```
void assertEqual(string expected, string actual, string message = "") {
    if (expected == actual)
    {
        cout << "PASSED: " << message << endl;
    }
    else {
        cout << "\tFAILED: " << message << endl;
        cout << " Expected: [\n" << expected << "]\n";
        cout << " Actual : [\n" << actual << "]\n";
    }
}

//in main()
assertEqual("**\n**\n", "*\n*\n", "length: 2");
```

```
void testDrawSquare()
{
    string expected1 = "**\n**\n";
    string actual1 = drawSquare(2);
    assertEqual(expected1, actual1, "length: 2");

    string expected2 = "***\n***\n";
    string actual2 = drawSquare(3);
    assertEqual(expected2, actual2, "length: 3");
}
```

```
string drawSquare(int length)
{
    string result = "";
    for (int i=0;i<length;i++)
    {
        for (int j=0;j<length;j++)
        {
            result += "*";
        }
        result += "\n";
    }
    return result;
}
```

Recursion

- Recursive: repeating unto itself
- Recursive function: contains a call to itself
- Solving problems with recursion involves breaking the problem down into smaller pieces
- When breaking a task into subtasks, it may be the case that the subtask is a smaller version of the same task

Example: the factorial function - Recall: `x! = 1 * 2 * 3 * ... * x` - `x! = x * (x-1) * (x-2) * ... * 2 * 1`

```
//a loop
int x = 5, fact = 1;
for (int k=0;k<x;k++)
{
    fact *= k;
}
```

- From mathematics, recall we can create a recursive formula from a sequence
- Example: the arithmetic sequence: `5, 10, 15, 20, 25, 30, ...`
- `a(n) = a(n-1) + 5`
    - `n` is the sequence position
    - the special case being when `n = 1`, `a(1) = 5` (this is the "starting value")

```
a(4) = a(3) + 5
     = (a(2) + 5) + 5
     = ((a(1) + 5) + 5) + 5
     = ((5 + 5) + 5) + 5
     = 20
```

The base case - If we assume that we start the sequence at `n = 1` then we could write a defitiion for `a(n)` like this: 1. If n = 1, then return 5 2. Otherwise, return `a(n-1) + 5` - We have to know what the base case is, otherwise the recursion never end!

```
int series(int n)
{
    //the base case
    if (n <= 1)
    {
        return 5;
    }

    //the recursive case
    return series(n-1) + 5;
}

//return series(3) + 5
//return series(2) + 5 + 5
//return series(1) + 5 + 5 + 5
//return 5 + 5 + 5 + 5

//n=4: return series(3) + 5
//n=3: return series(2) + 5
//n=2: return series(1) + 5
//n=1: return 5
```

```
//x! = x * (x-1) * (x-2) * ... * 1
int factorial(int n)
{
    //the base case
    if (n <= 1)
    {
        return 1;
    }

    //the recursive case
    return n * factorial(n-1);
}
```

Example: vertical numbers - Problem: write a recursive function definition that takes a non-negative integer and prints it out one digit-at-a-time vertically.

vertical(3); 3 vertical(12); 1 2 vertical(123); 1 2 3

Analysis: - Take a decimal number like 543 - How to separate the digits? 543 = 500 + 40 + 3

Algorithm: - Simplest case: - If n is 1 digit long, just print it - > 1-digit case: 1. Output all but the last digit vertically (recursive) 2. Write the last (least-significant) digit (base case) - Step 1 is a smaller version of the original task - Step 2 is the simplest case, the base case

```
void vertical(int n);
//Precondition: n >= 0
//Postcondition: n is written to the screen vertically with each digit on a separate line

void vertical(int n)
{
    if (n < 10) //base case
    {
        cout << n << endl;
    }
    else //two or more digits
    {
        // n / 10 (int division) returns n with just the least significant digit removed
        // 85 / 10 = 8
        vertical(n / 10);
        // n % 10 returns the least significant digit
        // 124 % 10 = 4
        cout << n % 10 << endl;
    }
}
```

vertical(543) - (5) -> print 3 | (1) V vertical(54) - (4) -> print 4 | (2) V vertical(5) - (3) -> print 5

"Infinite" recursion - A function that never reaches the base case, in theory, will run forever - If we wrote vertical() without the base case, then - eventually it will call vertical(0) - which will call vertical(0) - which will call vertical(0) - ... - In practice, the program will run out of memory and terminate abnormally

Stacks for recursion

- Computers use a memory structure called a stack to keep track of recursion
- Analogy: a stack of "paper"
    - Start at zero: no papers
    - To place data on the stack: write it on a piece of paper and place it on top of the stack
    - Insert more information: new paper, place it on top of the stack
    - To retrieve information: you can only take the top sheet of paper (then you throw it away when you're done reading it)
    - If you want to access any paper farther down, you go through the stack to get to it

LIFO (Last-in first-out): - When we put data in a LIFO, we call it a push - When we retrieve from a LIFO, we call it a pop - The other common scheme in data organization is FIFO (first-in first-out) AKA queue.

Stacks and making recursive calls - When execution of a function definition reaches a recursive call 1. Execution of this function is paused 2. Data is then saved in a new place on top of the stack (in memory) 3. Then, a new place in memory is "prepared" for the recursive call - A new function definition is written, arguments are plugged in - Execution of the recursive call starts 4. New data is saved on top of the stack 5. Repeat until we reach the base case

```
Stack
=====
vertical(5) // base case
vertical(54)
vertical(543)
```

When a recursive call gets to the base case 1. The computer retrieves the top memory unit of the stack 2. It resumes computation based on the information in that stack 3. When that computation ends, that memory unit is discarded 4. The next memory unit on top of the stack is retrieved so that processing can resume 5. The process continues until the stack is empty

Stack overflow (not the website) - Stacks are finite objects - Infinite recursions can force the stack to grow beyond its physical limits - The result is called a stack overflow. This causes abnormal termination

Other recursive functions - Recursive functions can have any return type (`void` , `int` , `string` , etc.)

A powers function - define a new power function (not the one in cmath) - let it return an integer 2^3 when we call the function as: `int y = power(2, 3);` - Use the following definition: `x^n = x^(n-1) * x` - E.g., 2^3 = 2^2 * 2 - Only works if n is positive

Tracing power(2, 3)

Stack

1 --> 1 power(2,0) * 2 --> 1 * 2 power(2,1) * 2 --> 2 * 2 power(2,2) * 2 --> 4 * 2 power(2,3) --> = 8

```
int power(int x, int n)
{
    //before the base case, it's a good idea to handle "illegal" input
    if (n < 0)
    {
        cerr << "Cannot use negative power!\n";
        return -1;
    }
    //base case
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return power(x, n-1) * x;
    }
}
```

Recusion vs. iteration - Any task that can be accomplished using recusrion can also be done without (using loops) and vice versa

```
int powerIterative(int x, int n)
{
    int p = 1;
    for (int k=1;k<=n;k++)
    {
        p *= x;
    }
    return p;
}
```

Reversing a string

```
string reverseStringIt(string s)
{
    string r = "";
    for (int i=s.length()-1;i>=0;i--)
    {
        r += s[i];
    }
    return r;
}

string reverseString(string s)
{
    //base case
    if (s.length() <= 1)
    {
        return s;
    }
    return s[s.length()-1] + reverseString(s.substr(0, s.length()-1));
}
```

Summing an integer array

```
int array[3] = {10, 20, 30};
int size = 3, sum = 0;
for (int i=0;i<size;i++)
{
    sum += array[i];
}
```

1. Recursive function needs two arguments: array, the size of the array
2. Repeat (recurse) with a smaller array (size - 1), then add its result to `array[size-1]` --> Push values onto the last stack.
3. Stop? When size is 0; sum of empty array is 0. The operations will be popped off the stack in this order: 0 + array[0] + array[1] + array[2] + ... + array[size-1]

```
int sumArray(int a[], int size)
{
    //base case
    if (size == 0)
    {
        return 0;
    }
    //recursive case
    return a[size-1] + sumArray(a, size-1);
}
```