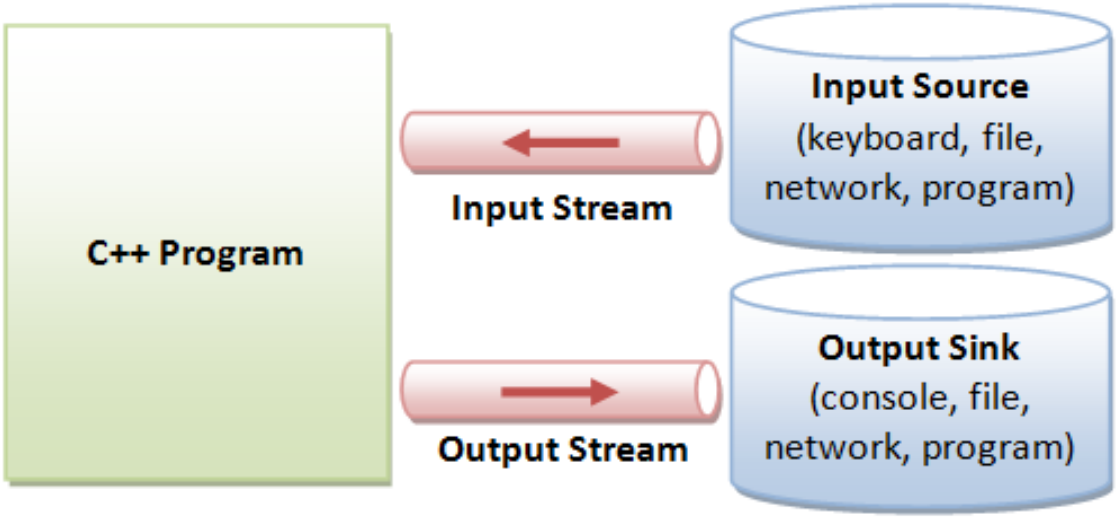
C++ Programming

Instructor: Rita Kuo
Office: CS 520E
Phone: Ext. 4405
E-mail: rita.kuo@uvu.edu

Streams

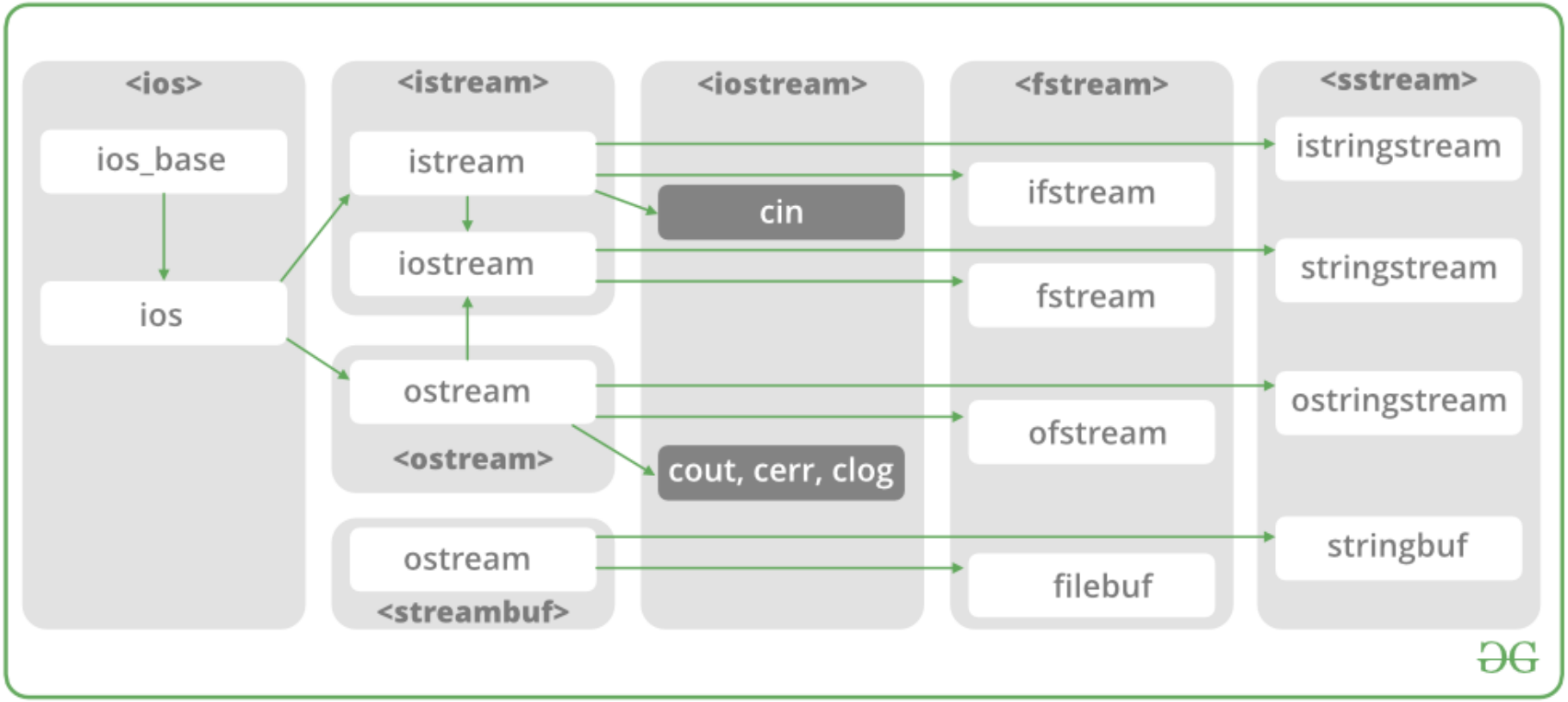
Streams

- C++ I/O occurs in streams
  - Sequence of bytes
  - Input: the bytes flow from a device (e.g., a keyboard, a disk drive, a net work connection, etc.) to main memory
  - Output: the bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, etc.)



https://www.quora.com/What-are-the-streams-in-C++

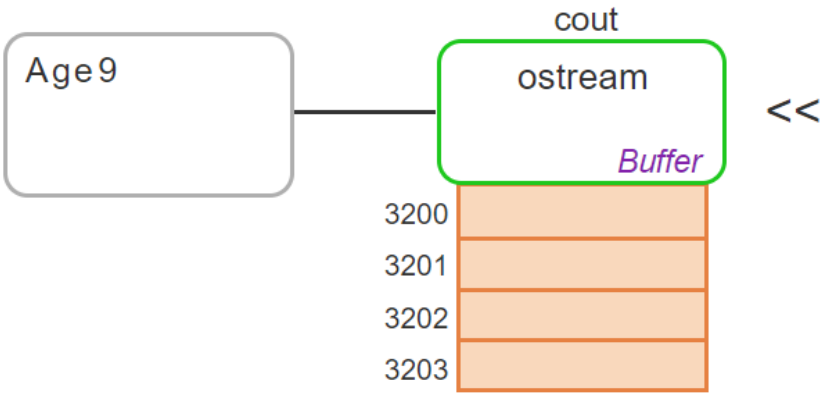
Streams



https://www.geeksforgeeks.org/file-handling-c-classes/

Streams

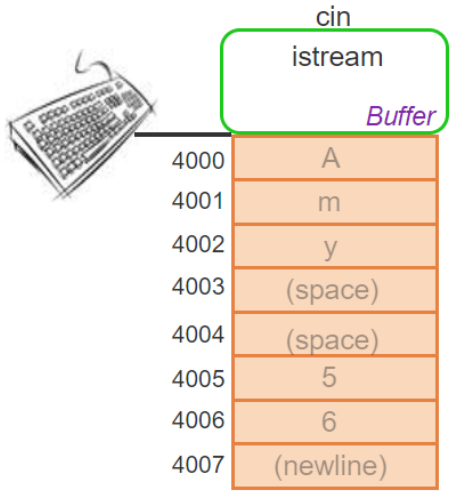
- The `ostream` (output stream) class
  - Supports output, available via `#include <iostream>`
  - Provides the `<<` operator, known as the insertion operator
    - Converts different types of data into a sequence of characters
      → The sequence is normally placed into a buffer, and the system then outputs the buffer via various times
    - Returns a reference to the `ostream` that called the operator
    - Is evaluated from left to right
    - Is overloaded with functions to support the various standard data types
- `cout`: A predefined `ostream` object




```
cout << "Age";
cout << ageYears;
```

Streams

- The `istream` (input stream) class
  - Supports input, available via `#include <iostream>`
  - Provides the `>>` operator, known as the extraction operator
    - Extract data from a data buffer
    - Write the data into different types of variables
    - Skips leading white spaces and extracts as many characters as possible consistent with the target variable types
- `cin`: a predefine `istream` object
  - Pre-associated with a system's standard input, usually a computer keyboard




```
cin >> firstName;
cin >> studentId;
```

Stream Manipulators

- Stream Manipulators
  - Perform formatting tasks, such as setting field widths, setting precisions, etc.
  - Overloads the insertion operator `<<` or extraction operator `>>` to adjust the way output appears
  - Most of the manipulators require the inclusion of the `<iomanip>` header
- Introduced manipulators
  - Floating-point manipulators
  - Text-alignment manipulators
  - Buffer manipulators
- Complete list of the stream manipulators
  - https://cplusplus.com/reference/library/manipulators/

Stream Manipulators

- Floating-point manipulators

Manipulator	Description	Example
fixed	Use fixed-point notation. From <iostream>	`// 12.340000` `cout << fixed << 12.34;`
scientific	Use scientific notation. From <iostream>	`// 1.234000e+01` `cout << scientific << 12.34;`

setprecision(p)	If stream has not been manipulated to fixed or scientific:
	Sets max number of digits in number
		`// 12.3`
		`cout << setprecision(3) << 12.34;`
		
		`// 12.34`
		`cout << setprecision(5) << 12.34;`

	If stream has been manipulated to fixed or scientific:
	Sets max number of digits in fraction only (after the decimal point).
	From <iomanip>
		`// 12.3`
		`cout << fixed << setprecision(1) << 12.34;`
		
		`// 1.2e+01`
		`cout << scientific << setprecision(1) << 12.34;`

showpoint	Even if fraction is 0, show decimal point and trailing 0s.
	Opposite is noshowpoint.
	From <iostream>
		`// 99`
		`cout << setprecision(3) << 99.0;`
		
		`// 99.0`
		`cout << setprecision(3) << showpoint << 99.0;`

Stream Manipulators

- Floating-point manipulators examples

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

int main() {
   double miles = 765.4261;

   cout << "setprecision(p) sets # digits" << endl;
   cout << miles << " (default p is 6)" << endl;
   cout << setprecision(8) << miles << " (p = 8)" << endl;
   cout << setprecision(5) << miles << " (p = 5)" << endl;
   cout << setprecision(2) << miles << " (p = 2)" << endl;
   cout << miles << endl << endl;

   // fixed uses fixed point notation
   cout << fixed;
   cout << "fixed: " << miles << endl;

   // scientific uses scientific notation
   cout << scientific;
   cout << "scientific: " << miles << endl;

   return 0;
}
```

```
setprecision(p) sets # digits
765.426 (default p is 6)
765.4261 (p = 8)
765.43 (p = 5)
7.7e+02 (p = 2)
7.7e+02

fixed: 765.43
scientific: 7.65e+02
```

Stream Manipulators

- Text-alignment manipulators

Manipulator	Description	Example

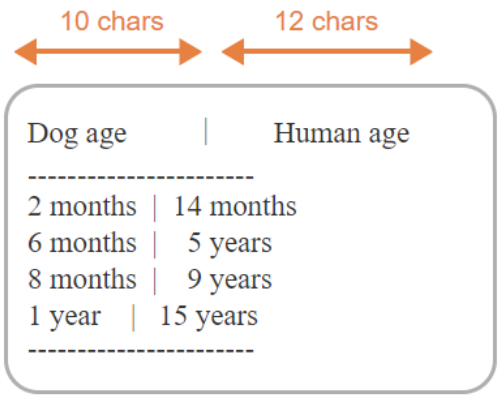
setw(n)	Sets the number of characters for the next output item only (does not persist, in contrast to other manipulators).
	By default, the item will be right-aligned, and filled with spaces.
	From <iomanip>
		`// "    Amy"`
		`// " George"`
		`cout << setw(7) << "Amy" << endl;`
		`cout << setw(7) << "George" << endl;`

setfill(c)	Sets the fill to character c. From <iomanip>	`// "****Amy"` `cout << setfill('*') << setw(7) << "Amy";`
left	Changes to left alignment. From <iostream>	`// "Amy    "` `cout << left << setw(7) << "Amy";`
right	Changes back to right alignment. From <iostream>	`// "    Amy"` `cout << right << setw(7) << "Amy";`

Stream Manipulators

- Text-alignment manipulators examples

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

int main() {
   // Dog age in human years (dogyears.com)
   cout << setw(10) << left  << "Dog age" << "|";
   cout << setw(12) << right << "Human age" << endl;

   // Produce long line
   cout << setfill('-') << setw(23) << "" << endl;

   // Reset fill character back to space
   cout << setfill(' ');

   cout << setw(10) << left  << "2 months" << "|";
   cout << setw(12) << right << "14 months" << endl;
   cout << setw(10) << left  << "6 months" << "|";
   cout << setw(12) << right << "5 years" << endl;
   cout << setw(10) << left  << "8 months" << "|";
   cout << setw(12) << right << "9 years" << endl;
   cout << setw(10) << left  << "1 year" << "|";
   cout << setw(12) << right << "15 years" << endl;

   // Produce long line
   cout << setfill('-') << setw(23) << "" << endl;

   return 0;
}
```



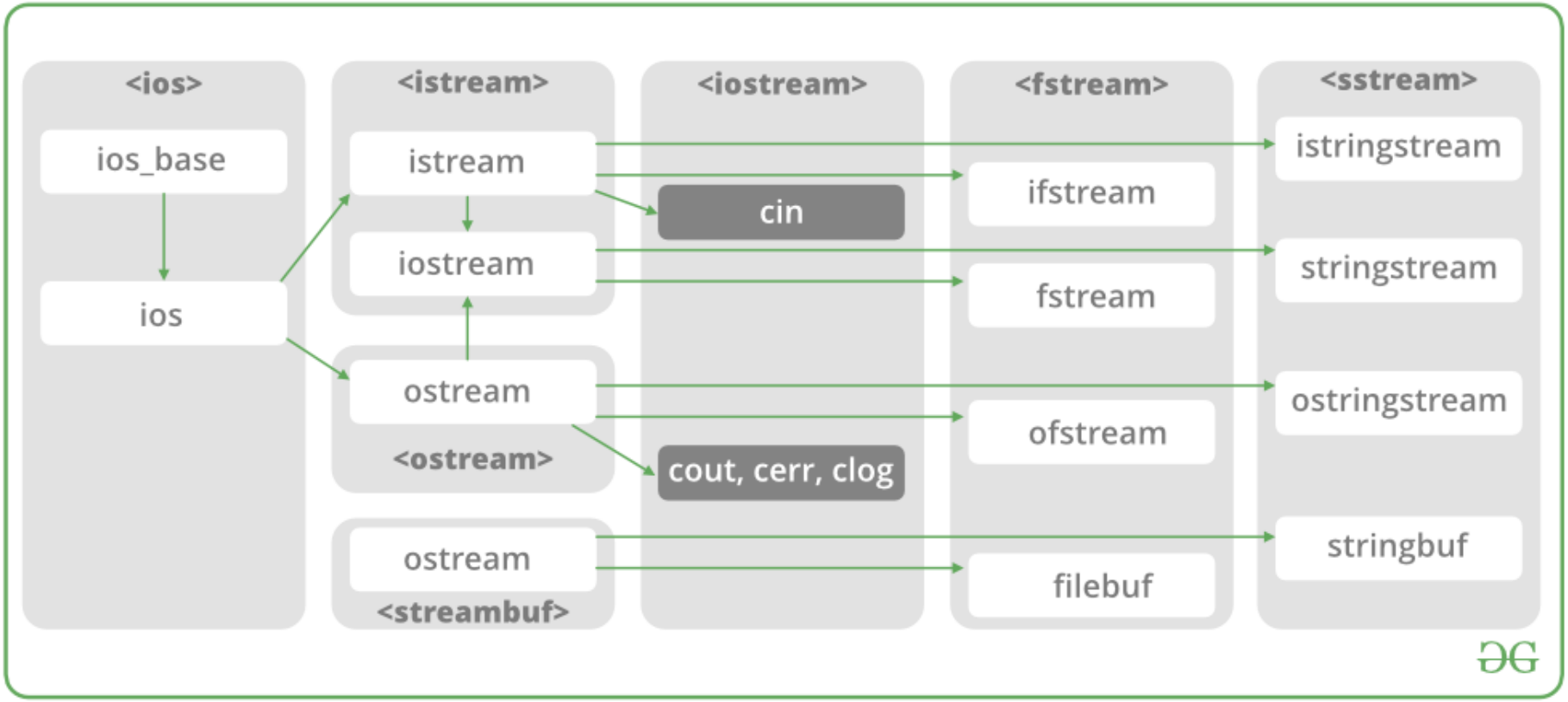
Stream Manipulators

- Buffer manipulators
  - Printing characters from the buffer to the output device (e.g., screen) requires a time-consuming reservation of processor resources
  - Once the resources are reserved, moving characters is fast, whether there is 1 character or 50 characters to print.
  - To preserve resources, the system may wait until the output buffer is full, or at least has a certain number of characters, before moving the characters to the output device
  - `endl` and `flush`: send all buffer contents to the output device without waiting

Manipulator	Description	Example
endl	Inserts a newline character '\n' into the output buffer and informs the system to flush the buffer. From <iostream>	`// Insert newline and flush` `cout << endl;`
flush	Informs the system to flush the buffer. From <iostream>	`// Flush buffer` `cout << flush;`

String Streams

Review - Streams



https://www.geeksforgeeks.org/file-handling-c-classes/

String Streams Processing

- String Stream Processing
  - The capabilities for inputting from, and outputting to, strings in memory
  - The capabilities are referred to as in-memory I/O or string stream processing
  - Should include the `<sstream>` and `<iostream>` headers
    - `istringstream` class: input from a string
    - `ostringstream` class: ouput to a string
- Usage of string stream processing
  - Data validation:
    - Read an entire line at a time from the input stream into a `string`
    - A validation routine can scrutinize the contents of the `string` and correct (or repair) the data, if necessary
  - Output formatting
    - Data can be prepared in a `string` to mimic the edited screen format
    - The `string` could be written to a disk file to preserve the screen image

Input String Stream

- An `istringstream` object inputs data from a string in memory to program variables

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


int main() {
   string  userInfo = "Amy Smith 19"; // Input string
   istringstream inSS(userInfo);      // Input string stream
   string firstName;                  // First name
   string lastName;                   // Last name
   int userAge;                       // Age

   // Parse name and age values from input string
   inSS >> firstName;
   inSS >> lastName;
   inSS >> userAge;


   // Output parsed values
   cout << "First name: " << firstName << endl;
   cout << "Last  name: " << lastName << endl;
   cout << "Age: "        << userAge << endl;

   return 0;
}
```

Input String Stream

- Using `getline()` with string streams
  - Process user input line-by-line

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

int main() {
   istringstream inSS;       // Input string stream
   string lineString;        // Holds line of text
   bool   inputDone = false; // Flag to indicate next iteration
   ...

   while (!inputDone) {

      // Entire line into lineString
      getline(cin, lineString);

      // Copies to inSS's string buffer
      inSS.clear();
      inSS.str(lineString);

      // Now process the line
   }
   return 0;
}
```

Input String Stream

- Reaching the end of a string stream
  - Input streams have a Boolean function called `eof()` or end of file that returns `true` or `false` depending on whether or not the end of the stream has been reached
  - Check if the end of input string stream has been reached with if or while statements

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

int main() {
   istringstream inSS;       // Input string stream
   string lineString;        // Holds input string
   string data;

   cout << "Enter a list of names separated by spaces: ";

   // Entire line into lineString
   getline(cin, lineString);
   inSS.str(lineString);

   while (inSS >> data) {
      cout << data << endl;
   }

   return 0;
}
```

Input String Stream

- Example: Phone number formats

```
// Try extracting area code.
inSS >> areaCode;
if (inSS.good()) {
   // Number format should be ###-###-####
   inSS >> dummyChar1 >> officeCode >> dummyChar2 >> stationNum;

   if (inSS.eof() && dummyChar1 == '-' && dummyChar2 == '-') {
      isValidNumber = true;
   }
}
else {
   // Number format should be (###) ###-####

   // Clear inSS state, and try extracting with area code in ()
   inSS.clear();
   inSS >> dummyChar1 >> areaCode >> dummyChar2;
   if (inSS.good()  && dummyChar1 == '(' && dummyChar2 == ')') {
      // Extract office code, then -, and then station number
      inSS >> officeCode >> dummyChar1 >> stationNum;
      if (inSS.eof() && dummyChar1 == '-') {
         isValidNumber = true;
      }
   }
}
```

```
Enter a 10-digit phone number (or -1 to exit):
342-555-9084
   Standardized format: (342) 555-9084

*1778.555.2925
   Invalid phone number.

778.555.2925
   Invalid phone number.

(302)555-8927
   Standardized format: (302) 555-8927

-1
```

Output String Stream

- An output string stream variable of type `ostringstream` can insert characters into a string buffer instead of the screen
  - A program can insert characters into an `ostringstream` buffer using `<<`, just like the `cout` stream
  - The `ostringstream` member function `str()` returns the contents of an `ostringstream` buffer as a string

Output String Stream

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

int main() {
   ostringstream infoOSS;     // Output string stream
   string infoStr;            // Information string
   string firstName;          // First name
   string lastName;           // Last name
   int userAge;               // Age

   // Prompt user for input
   cout << "Enter \"firstname lastname age\": " << endl;
   cin >> firstName;
   cin >> lastName;
   cin >> userAge;

   // Write user input to string stream
   infoOSS << lastName << ", " << firstName;
   infoOSS << " " << userAge;

   // Appends (minor) to string stream if less than 21
   if (userAge < 21) {
      infoOSS << " (minor)";
   }

   // Extract string stream buffer as a single string
   infoStr = infoOSS.str();

   cout << "Information: " << infoStr << endl;

   return 0;
}
```

Output String Stream

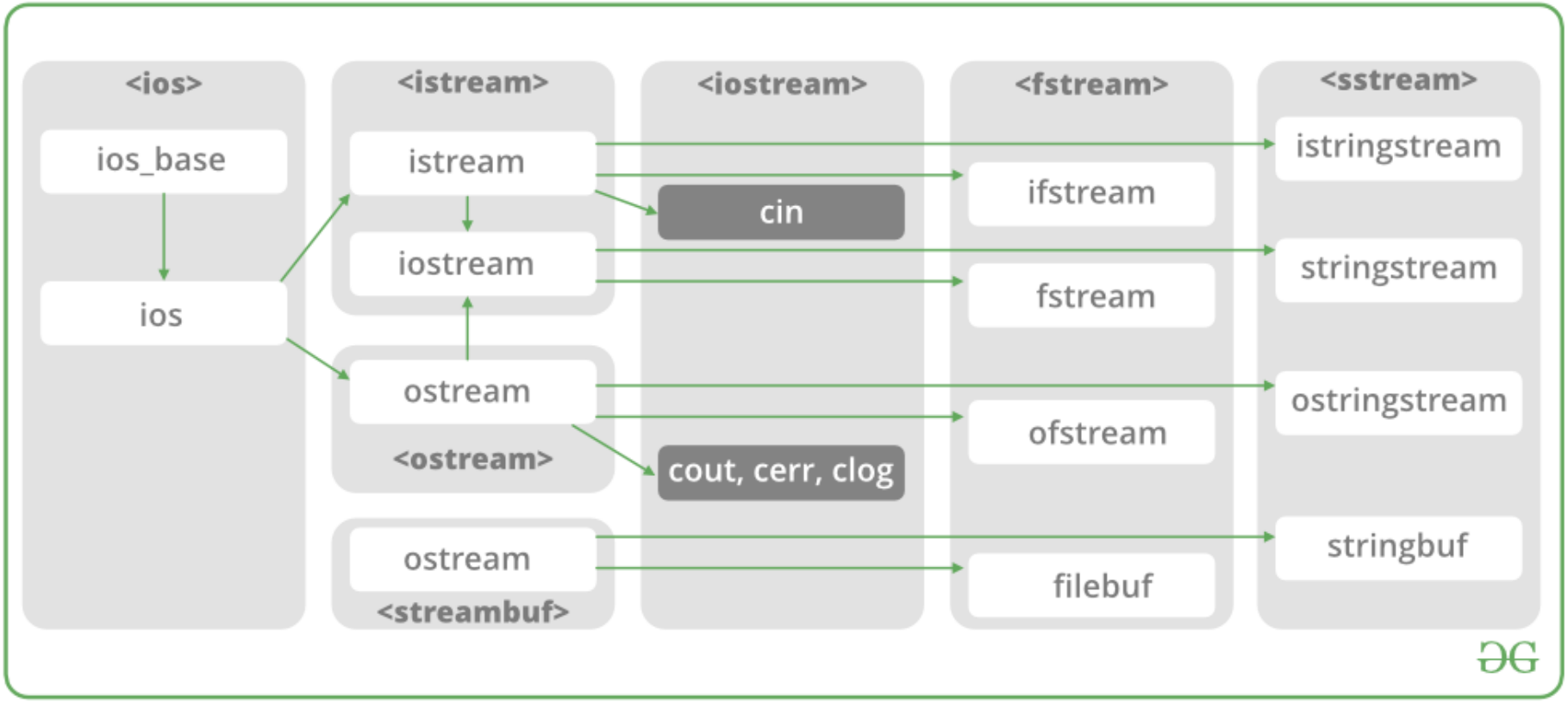
- Example: Savings table

```
string ProduceSavingsTable(double startAmount, double apr, int numYears) {
   // Column widths
   const int YEAR_COL_WIDTH = 5;
   const int BALANCE_COL_WIDTH = 10;

   ostringstream outSS;
   double interest;
   double balance = startAmount;
   int month;
   int totalMonths = numYears * 12;

   // Convert APR to monthly percentage rate and decimal number
   double mpr = apr / 12 * 0.01;

   // Display 2 decimal places
   outSS << fixed << setprecision(2);

   // Table heading
   outSS << setw(YEAR_COL_WIDTH) << "Year"
      << setw(BALANCE_COL_WIDTH) << "Balance" << endl;

   // Calculate interest and ending balance for each month
   for (month = 1; month <= totalMonths; ++month) {
      interest = balance * mpr;
      balance += interest;

      // Only output year number and balance at the end of the year
      if (month % 12 == 0) {
         outSS << setw(YEAR_COL_WIDTH) << month / 12
             << setw(BALANCE_COL_WIDTH) << balance << endl;
      }
   }

   // Return the table as a string
   return outSS.str();
}
```

```
Starting amount?
100
Annual Percentage Rate?
5
Number of years?
6

Savings over time:
 Year   Balance
    1    105.12
    2    110.49
    3    116.15
    4    122.09
    5    128.34
    6    134.90
```

File I/O

Review - Streams



https://www.geeksforgeeks.org/file-handling-c-classes/

Files and Streams

- Computers store files on secondary storage devices
- C++ views each file simply as a sequence of bytes
- Each file ends either
  - With and end-of-file marker, or
  - At a specific byte number recorded in an operating system-maintained, administrative data structure
    - When a file is opened, an object is created
    - A stream is associated with the object
- `<iostream>` and `<fstream>` must be included
  - `basic_ifstream`: for file input
  - `basic_ofstream`: for file outputs
  - `basic_fstream`: for file input and output

Files Input

- Opening and reading from a file
  - Create a new input stream that comes from a file
    `ifstreamInstance.open(filePath)`
  - Read data from the opened file like the `cin` stream
    `ifstreamInstance >> variable`
  - Close the opened file
    `ifstreamInstance.close()`
- Reading until the end of the file
  - A program can read varying amounts of data in a file by using a loop that reads until the end of the file has been reached.
  - The `eof()` function returns `true` if the previous stream operation reached the end of the file.
  - Errors may be encountered while attempting to read from a file
    → The `fail()` function returns `true` if the previous stream operation had an error

Files Input

- Opening and reading from a file example

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

int main() {
   ifstream inFS;      // Input file stream
   int fileNum1;       // Data value from file
   int fileNum2;       // Data value from file

   // Try to open file
   cout << "Opening file numFile.txt." << endl;

   inFS.open("numFile.txt");
   if (!inFS.is_open()) {
      cout << "Could not open file numFile.txt." << endl;
      return 1; // 1 indicates error
   }

   // Can now use inFS stream like cin stream
   // numFile.txt should contain two integers, else problems
   cout << "Reading two integers." << endl;
   inFS >> fileNum1;
   inFS >> fileNum2;
   cout << "Closing file numFile.txt." << endl;
   inFS.close(); // Done with file, so close it

   // Output values read from file
   cout << "num1: " << fileNum1 << endl;
   cout << "num2: " << fileNum2 << endl;
   cout << "num1 + num2: " << (fileNum1 + fileNum2) << endl;

   return 0;
}
```

numFile.txt with two integers:

```
5
10
```

Failure to open file

```
Opening file numFile.txt.
Could not open file numFile.txt.
```

Successfully open file

```
Opening file numFile.txt.
Reading two integers.
Closing file numFile.txt.
num1: 5
num2: 10
num1 + num2: 15
```

Files Input

- Reading until the end of the file example

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

int main() {
   ifstream inFS;   // Input file stream
   int fileNum;     // File data

   // Open file
   cout << "Opening file myfile.txt." << endl;
   inFS.open("myfile.txt");

   if (!inFS.is_open()) {
      cout << "Could not open file myfile.txt." << endl;
      return 1;
   }

   // Print read numbers to output
   cout << "Reading and printing numbers." << endl;

   inFS >> fileNum;
   while (!inFS.fail()) {
      cout << "num: " << fileNum << endl;
      inFS >> fileNum;
   }
   if (!inFS.eof()) {
      cout << "Input failure before reaching end of file." << endl;
   }

   cout << "Closing file myfile.txt." << endl;

   // Done with file, so close it
   inFS.close();

   return 0;
}
```

Files Input

- Example: Counting instances of a specific word

```
ifstream inFS;     // Input file stream
string userWord;
int wordFreq = 0;
string currWord;

// Open file
cout << "Opening file wordFile.txt." << endl;
inFS.open("wordFile.txt");

if (!inFS.is_open()) {
   cout << "Could not open file wordFile.txt." << endl;
   return 1;
}

// Word to be found
cout << "Enter a word: ";
cin >> userWord;

// Identify when a word matches the userWord
// and increase wordFreq
while (!inFS.eof()) {
   inFS >> currWord;
   if (!inFS.fail()) {
      if(currWord == userWord) {
         ++wordFreq;
      }
   }
}
```

Files Input

- Example: Business reviews

```
void ReadReviews(string& restaurantName, vector<string>& userNames,
                 vector<int>& userRatings) {
   ifstream inFS;     // Input file stream
   string userName;
   int userRating;

   // Open file
   inFS.open("Trattoria_Reviews.txt");

   if (!inFS.is_open()) {
      cout << "Could not open file Trattoria_Reviews.txt."<< endl;
      return;
   }

   getline(inFS, restaurantName);

   while (!inFS.eof()) {
      inFS >> userName;
      inFS >> userRating;

      if (!inFS.fail()) {
         userNames.push_back(userName);
         userRatings.push_back(userRating);
      }
   }

   // Close file when done reading
   inFS.close();
}
```

Input stream errors

- The stream entering an error state
  - Insertion or extraction fails
    Example: If a file has the string `two` but the program attempts to extract an integer
  - A value extracted is too large (or small) to fit the given variable
    → the input stream may skip extraction, set the given variable to 0, or set the given variable to the maximum (or minimum) value of the variable's data type
- Stream error state flags
  - Several 1-bit error flags → track the state of the stream

Flag	Meaning	Function
goodbit	Indicates no error flags are set and the stream is good.	***good()*** returns true if no stream errors have occurred.
eofbit	Indicates if end-of-file reached on extraction.	***eof()*** returns value of eofbit, if end-of-file reached on extraction.
failbit	Indicates a logical error for the previous extraction or insertion operation.	***fail()*** returns true if either failbit or badbit is set, indicating an error for the previous stream operation.
badbit	Indicates an error occurred while reading or writing the stream, and the stream is bad. Further operations on the stream will fail.	***bad()*** returns true if badbit is set, indicating the stream is bad.

Input stream errors

- Check for errors while reading a file

```
inFS.open("Reviews.txt");

if (!inFS.is_open()) {
   cout << "Could not open file Reviews.txt."<< endl;
}

while (!inFS.eof() && inFS.good()) {
   inFS >> userName;
   inFS >> userRating;

   if (!inFS.fail()) {
      userNames.push_back(userName);
      userRatings.push_back(userRating);
   }
}

// If end-of-file not reached, then an error occurred
if (!inFS.eof()) {
   cout << "Error reading Reviews.txt." << endl;
   exit(EXIT_FAILURE);
}
```

Reviews.txt

lazydog28 ✓
5 ✓

Vancity93 ✓ string
three ✗ int

stun7ning
4

inFS state bits

0	goodbit
0	eofbit
1	failbit
0	badbit

Error reading Reviews.txt.

File output

- Opening and writing a file

Action	Sample code
Open the file helloWorld.txt for writing	`ofstream outFS;` `outFS.open("helloWorld.txt");`

Check to see if the file opened successfully	`if (!outFS.is_open()) {`
	`  // Do not proceed to code that writes to the file`
	`}`

Write the string "Hello World!" to the file	`outFS << "Hello World!" << endl;`
Close the file after writing all desired data	`outFS.close();`

File output

- Example: Writing a text file

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

int main() {
   ofstream outFS; // Output file stream

   // Open file
   outFS.open("myoutfile.txt");

   if (!outFS.is_open()) {
      cout << "Could not open file myoutfile.txt." << endl;
      return 1;
   }

   // Write to file
   outFS << "Hello" << endl;
   outFS << "1 2 3" << endl;

   // Done with file, so close
   outFS.close();

   return 0;
}
```

Contents of myoutfile.txt:

```
Hello
1 2 3
```

File output

- Example: Writing a simple HTML file

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

void writeHTMLFile(ostream& outStream, string innerHTML) {
   outStream << "<!DOCTYPE html>" << endl;
   outStream << "<html>" << endl;
   outStream << "  <body>" << endl;
   outStream << "    <p>" << innerHTML << "</p>" << endl;
   outStream << "  </body>" << endl;
   outStream << "</html>" << endl;
}

int main() {
   string htmlParagraph = "Hello <b>HTML</b> world!";

   // Open an output file stream
   ofstream outFS;
   outFS.open("simple.html");

   if (!outFS.is_open()) {
      cout << "Could not open file simple.html." << endl;
      return 1;
   }

   // Write to, and then close, file
   writeHTMLFile(outFS, htmlParagraph);
   outFS.close();

   // Use the same function, writeHTMLFile, to write to cout
   writeHTMLFile(cout, htmlParagraph);
   return 0;
}
```

Console:

```
<!DOCTYPE html>
<html>
 <body>
   <p>Hello <b>HTML</b> world!</p>
 </body>
</html>
```

simple.html file contents:

```
<!DOCTYPE html>
<html>
 <body>
   <p>Hello <b>HTML</b> world!</p>
 </body>
</html>
```

Command-Line Arguments

Array of C Strings

- How to store an array of strings?
  - A two-dimensional array of characters

```
char planets[][8] = {"Mercury", "Venus", "Earth", "Mars", "Jupiter",
                     "Saturn", "Uranus", Neptune", Pluto"};
```

  - Not all strings were long enough to fill an entire row of the array
  - Padded them with null characters → wasted space
  - Most collections of strings will have a mixture of long strings and short strings
  - We need a ragged array: a two-dimensional array whose rows can have different lengths

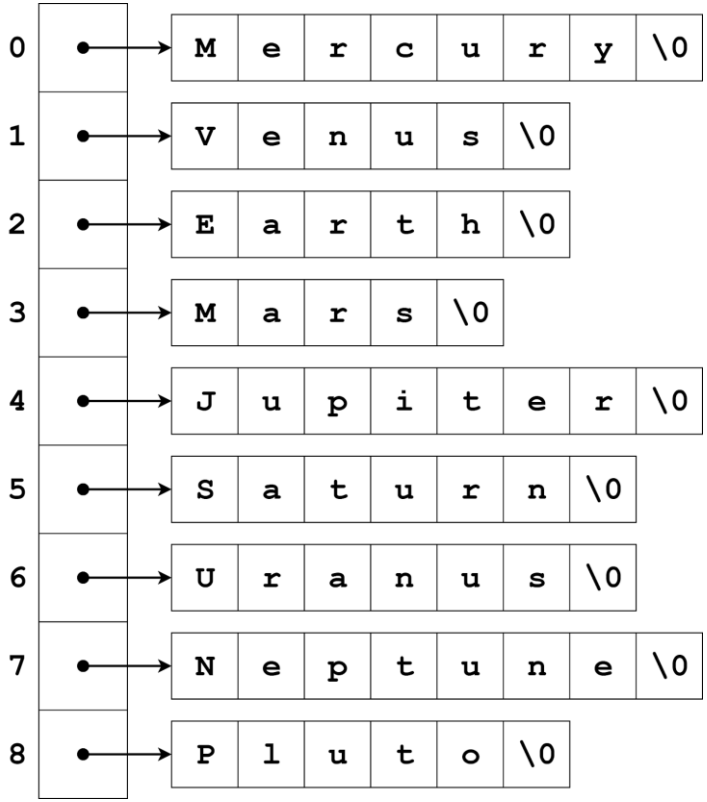
	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

Array of C Strings

- How to store an array of strings?
  - An array whose elements are points to strings

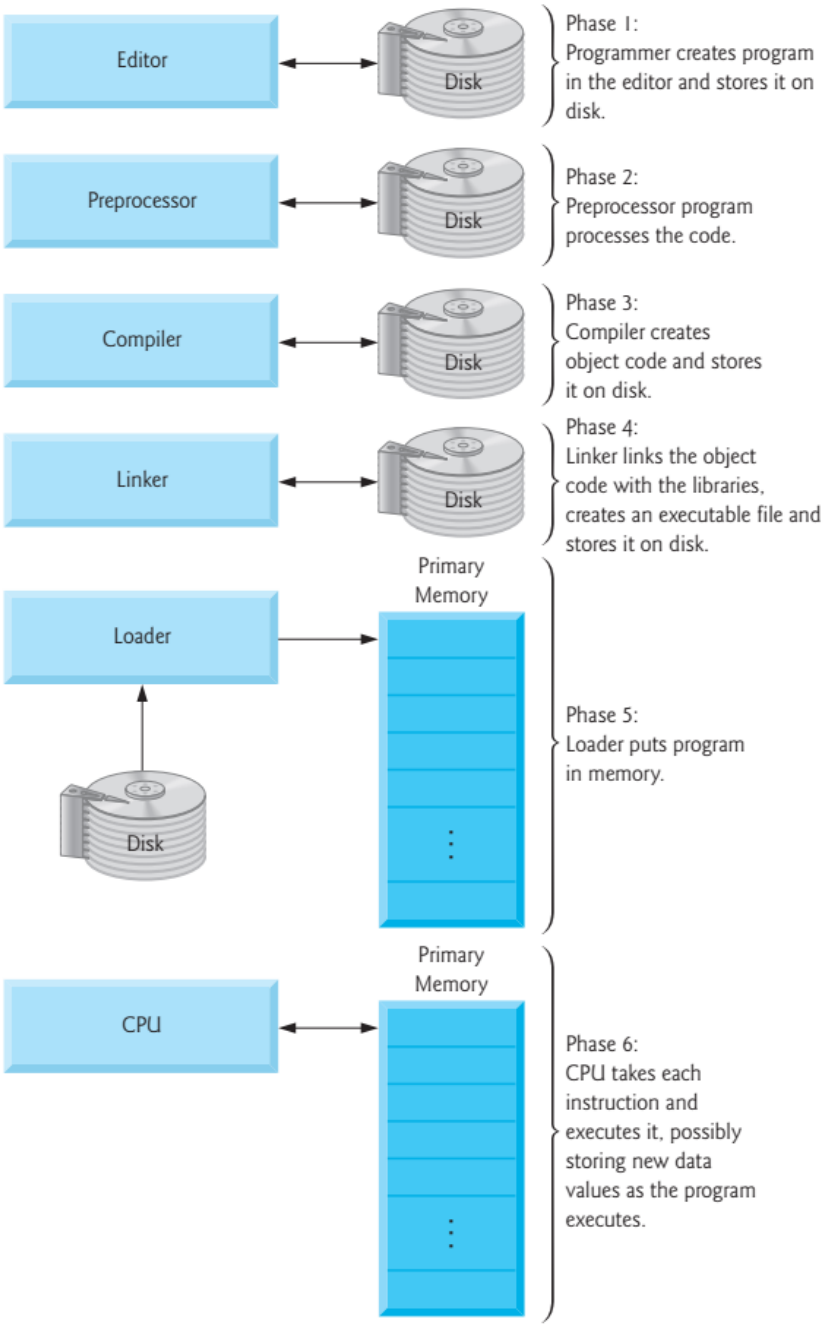
```
char *planets[] = { "Mercury", "Venus", "Earth", "Mars", "Jupiter",
                    "Saturn", "Uranus", Neptune", Pluto"};
```

  - Each element of `planets` is a pointer to a null-terminated string
  - There are no longer any wasted characters in the strings.



Review – C++ Working Environment

- A text-editor - write source code
  - VSCode, Atom, etc.
- A compiler - translate source code into machine language
  - Linux: GNU C++ Compiler
  - Windows: Visual Studio, MinGW
  - Mac: Xcode
- A shell - a way to interact with the kernel; a means to execute the program (Unix)



Linux Environment

- Basic Linux Commands
  - Format: `command - options arguments`

```
rita@CSE113: ~
File  Edit  View  Search  Terminal  Help
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

rita@CSE113:~$ pwd
/home/rita
rita@CSE113:~$ ls
Desktop    Downloads         Music     Public     Videos
Documents  examples.desktop  Pictures  Templates
rita@CSE113:~$ mkdir CSE113
rita@CSE113:~$ ls
CSE113   Documents  examples.desktop  Pictures  Templates
Desktop  Downloads  Music             Public    Videos
rita@CSE113:~$ cd CSE113
rita@CSE113:~/CSE113$ pwd
/home/rita/CSE113
rita@CSE113:~/CSE113$ ls
rita@CSE113:~/CSE113$ touch readme.txt
rita@CSE113:~/CSE113$ ls
readme.txt
rita@CSE113:~/CSE113$ cd ..
rita@CSE113:~$ pwd
/home/rita
rita@CSE113:~$
```

`pwd`: print current working directory
`ls`: list files in the current directory
`mkdir`: make directory
`cd`: change working directory to
`touch`: create new, empty files
`.`: current directory
`..`: parent directory
`/`: root, everything starts in this directory
`~`: user's home directory

Linux Environment

- Compiling hello.c in terminal again

```
File  Edit  View  Search  Terminal  Help
rita@CSE113:~/CSE113/exercise$ gcc -g -Wall hello.c
rita@CSE113:~/CSE113/exercise$ ls
a.out  hello.c
rita@CSE113:~/CSE113/exercise$
```

  - `-g`: Produce debugging information in the operating system's native format
  - `-Wall`: Enables all the warnings about constructions that some users consider questionable, and that are easy to avoid, even in conjunction with macro
  - `a.out`: the name of the executable file that was created when you compiled your code
  - `-o`: change the name of the executable file that you create

```
File  Edit  View  Search  Terminal  Help
rita@CSE113:~/CSE113/exercise$ gcc -g -Wall hello.c -o hello
rita@CSE113:~/CSE113/exercise$ ls
a.out  hello  hello.c
rita@CSE113:~/CSE113/exercise$
```

https://gcc.gnu.org/onlinedocs/gcc/Option-Index.html

Linux Environment

- Execute the executable file
  - Type `hello` in the exercise directory

```
File  Edit  View  Search  Terminal  Help
rita@CSE113:~/CSE113/exercise$ ls
a.out  hello  hello.c  hello.o  hello.s
rita@CSE113:~/CSE113/exercise$ hello

Command 'hello' not found, but can be installed with:

sudo apt install hello
sudo apt install hello-traditional

rita@CSE113:~/CSE113/exercise$
```

  - PATH environment variable:
  - How the system searches for executable program

```
File  Edit  View  Search  Terminal  Help
rita@CSE113:~/CSE113/exercise$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/lo
cal/games:/snap/bin
rita@CSE113:~/CSE113/exercise$
```

`~/CSE113/exercise` is not in the `PATH` variable

Linux Environment

- Execute the file
  - Type `./hello` instead

```
File  Edit  View  Search  Terminal  Help
rita@CSE113:~/CSE113/exercise$ ./hello
Hello, world!
rita@CSE113:~/CSE113/exercise$
```

  - Need to tell the shell where the file is in the file system
  - `./`: tell the shell execute the file in this directory

Command-Line Arguments

- Program parameters in C/C++
  - Define `argc` and `argv` parameters in main function

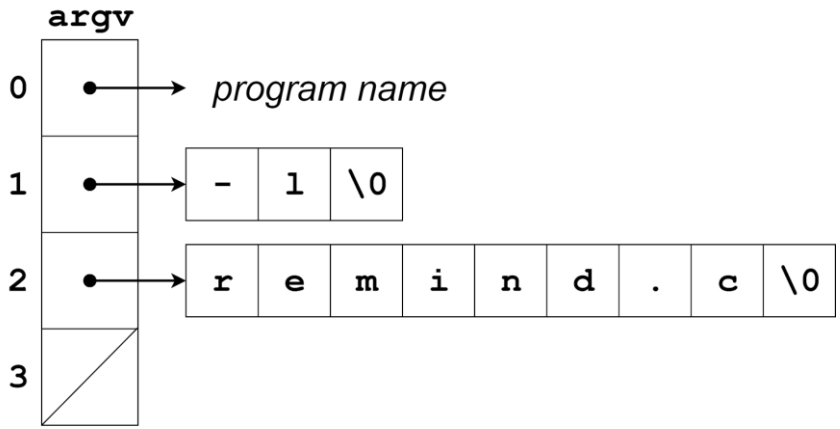
```
int main(int argc, char *argv[])
{ ... }
```

  - `argc` (argument count): the number of command-line arguments
  - `argv` (argument vector): an array of pointers to the command-line arguments, which are stored in string form
    - `argv[0]`: points to the name of the program
    - `argv[1]` through `argv[argc-1]`: points to the remaining command-line arguments
    - `argv[argc]`: always a null pointer → a special pointer that points to nothing

Command-Line Arguments

- Program parameters in C/C++
  - Example:

    `ls -l remind.c`



    - `argc`: 3
    - `argv[0]`: point to a string containing the program name. May include a path or other information that depends on the operating system.
    - `argv[1]`: point to the string `"-l"`
    - `argv[2]`: point to the string `"remind.c"`
    - `argv[3]`: a null pointer

Command-Line Arguments

- Example

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

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

   // Prints argc and argv values
   cout << "argc: " << argc << endl;
   for (i = 0; i < argc; ++i) {
      cout << "argv[" << i << "]: " << argv[i] << endl;
   }

   return 0;
}
```

```
> ./argtest
argc: 1
argv[0]: ./argtest

> ./argtest Hello
argc: 2
argv[0]: ./argtest
argv[1]: Hello

> ./argtest Hey ABC 99 -5
argc: 5
argv[0]: ./argtest
argv[1]: Hey
argv[2]: ABC
argv[3]: 99
argv[4]: -5
```

Command-Line Arguments

- Example

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

/* Usage: program username userage */
int main(int argc, char* argv[]) {
   string nameStr;  // User name
   string ageStr;   // User age

   // Get inputs from command line
   nameStr = argv[1];
   ageStr  = argv[2];

   // Output result
   cout << "Hello " << nameStr << ". ";
   cout << ageStr << " is a great age." << endl;

   return 0;
}
```

```
> myprog.exe Amy 12
Hello Amy. 12 is a great age.

...

> myprog.exe Rajeev 44 HEY
Hello Rajeev. 44 is a great age.

...

> myprog.exe Denming
Segmentation fault
```

Command-Line Arguments

- Example

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

/* Usage: program username userage */
int main(int argc, char* argv[]) {
   string nameStr;  // User name
   string ageStr;   // User age

   // Check if correct number of arguments provided
   if (argc != 3) {
      cout << "Usage: myprog.exe name age" << endl;
      return 1; // 1 indicates error
   }

   // Get inputs from command line
   nameStr = argv[1];
   ageStr = argv[2];

   // Output result
   cout << "Hello " << nameStr << ". ";
   cout << ageStr << " is a great age." << endl;

   return 0;
}
```

```
> myprog.exe Amy 12
Hello Amy. 12 is a great age.

...

> myprog.exe Denming
Usage: myprog.exe name age

...

> myprog.exe Alex 26 pizza
Usage: myprog.exe name age
```

Command-Line Arguments

- Example

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

int main(int argc, char* argv[]) {
   ifstream inFS;

   // Check number of arguments
   if (argc != 2) {
      cout << endl << "Usage: myprog.exe inputFileName" << endl;
      return 1;   // 1 indicates error
   }

   // Try to open file
   cout << "Opening file " << argv[1] << "." << endl;
   inFS.open(argv[1]);

   if (!inFS.is_open()) {
      cout << "Could not open file " << argv[1] << "." << endl;
      return 1;
   }

   ...

   inFS.close();

   return 0;
}
```

Stream Operator Overloading

Review - Function Signatures

- In regular function
  - The name and the parameter-type-list of a function
- In a class member
  - The name and the parameter-type-list of a function
  - The class, concept, concept map, or the namespace
- Functions in the same scope must have unique signatures
- The compiler uses to perform overload resolution

https://stackoverflow.com/questions/2322736/what-is-the-difference-between-function-declaration-and-signature

Review - Function Overloading

- Function Overloading
  - A program has two functions with the same name but differing in the number or types of parameters
- How the compiler differentiates overloaded function
  - By their signatures, combining with a function's name and its parameter types (in order)
  - Name mangling or name decoration
    Encodes each function identifier with the number and types of its parameters → enable type-safe linkage
  - Type-safe linkage: ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters

Review - Constructor Overloading

- Provide different initialization values when creating a new object
- Define multiple constructors differing in parameter types
- Example

```
class Restaurant {
   public:
      Restaurant();
      Restaurant(string initName, int initRating);

   ...
};

// Default constructor
Restaurant::Restaurant() {
   name = "NoName";
   rating = -1;
}

// Another constructor
Restaurant::Restaurant(string initName, int initRating) {
   name = initName;
   rating = initRating;
}

int main() {
   Restaurant foodPlace;              // Calls default constructor

   Restaurant coffeePlace("Joes", 5); // Calls another constructor

   ...
}
```

foodPlace

Name: NoName
Rating: -1

coffeePlace

Name: Joes
Rating: 5

Review - Operator Overloading

- Redefine the functionality of built-in operators like +, -, and *, to operate on programmer-defined objects

Without operator overloading

```
TimeHrMn time1(3, 22);
TimeHrMn time2(2, 50);
TimeHrMn timeTot;
timeTot.hours = time1.hours + time2.hours;
timeTot.minutes = time1.minutes + time2.minutes;

timeTot.Print();
```

Console:

```
H:5, M:72
```

With operator overloading

```
TimeHrMn time1(3, 22);
TimeHrMn time2(2, 50);
TimeHrMn timeTot;
timeTot = time1 + time2;

timeTot.Print();
```

Console:

```
H:5, M:72
```

- Example
  - <<: is used both as the stream insertion operator and as the bitwise left-shift operator

Review - Operator-Overloading Function

- Overload + operator as a non-member function and test the program again

TimeHrMn.h

```
class TimeHrMn {
  ...
};

TimeHrMn operator+(int lhs, const TimeHrMn& rhs);
bool operator==(const TimeHrMn& lhs, const TimeHrMn& rhs);
bool operator<(const TimeHrMn& lhs, const TimeHrMn& rhs);
```

TimeHrMn.cpp

```
TimeHrMn operator+(int hours, const TimeHrMn& rhs)
{
  TimeHrMn timeTotal(rhs.GetHours() + hours, rhs.GetMinutes());
  return timeTotal;
}
```

Review - Friend Functions

- A friend function of a class is defined outside that class's scope, yet has the right to access the non-`public` (and `public`) members of the class
- The `friend` declaration can be appear anywhere in the class
- Update the overloaded + operator non-member function as a friend function

TimeHrMn.h

```
class TimeHrMn {
  friend TimeHrMn operator+(int lhs, const TimeHrMn& rhs);
  ...
};

bool operator==(const TimeHrMn& lhs, const TimeHrMn& rhs);
bool operator<(const TimeHrMn& lhs, const TimeHrMn& rhs);
```

Overloading Stream Operators

- Overloading the << operator (insertion operator)

Figure 11.11.2

```
#include <iostream>
#include <queue>

class WaitingLine
{
public:
   WaitingLine& operator<<(const string& name) {
      // Add the name to the end of the line
      line.push(name);

      cout << name << " enters the back of the line" << endl;

      return *this;
   }
   ...
   queue<string> line;
};

int main() {
   WaitingLine line1;
   line1 << "Lion";
   line1 << "Tiger";
   line1 << "Bear";
   ...
   return 0;
}
```

Overloading Stream Operators

- Overloading the >> operator (extraction operator)

Figure 11.11.2

```
#include <iostream>
#include <queue>

class WaitingLine
{
public:
 WaitingLine& operator>>(string& frontName) {
      // Copy the name at the front of the line to frontName, then remove
      frontName = line.front();
      line.pop();

      return *this;
   }
   ...
   queue<string> line;
};

int main() {
   WaitingLine line1;
 ...
   string name;
   for (int i = 0; i < 2; i++) {
      line1 >> name;
      cout << name << " exits the front of the line" << endl;
   }   return 0;
}
```

Overloading Stream Operators

- Extending `cin` and `cout`

```
class WaitingLine
{
public:
   ...
   friend ostream& operator<<(ostream& out, const WaitingLine& line) {
      out << "(front)";
      queue<string> lineCopy = line.line;
      while (!lineCopy.empty()) {
         string lineItem = lineCopy.front();
         lineCopy.pop();
         out << " " << lineItem;
      }
      out << " (back)";
      return out;
   }
   friend istream& operator>>(istream& in, WaitingLine& line) {
      string inString;
      in >> inString;
      line << inString;
      return in;
   }
   queue<string> line;
};
```

It is much more common to overload `<<` and `>>` using non-member functions

Overloading Stream Operators

- Extending `cin` and `cout`

```
int main() {
   WaitingLine line1;

   // Get user input to add an item to the line
   cin >> line1;

   // Add a 2nd item to the line
   line1 << "Item_2";

   cout << "Line: " << line1 << endl;
   return 0;
}
```