C/C++ Review

Overview

Review, Recommendations and Gotchas

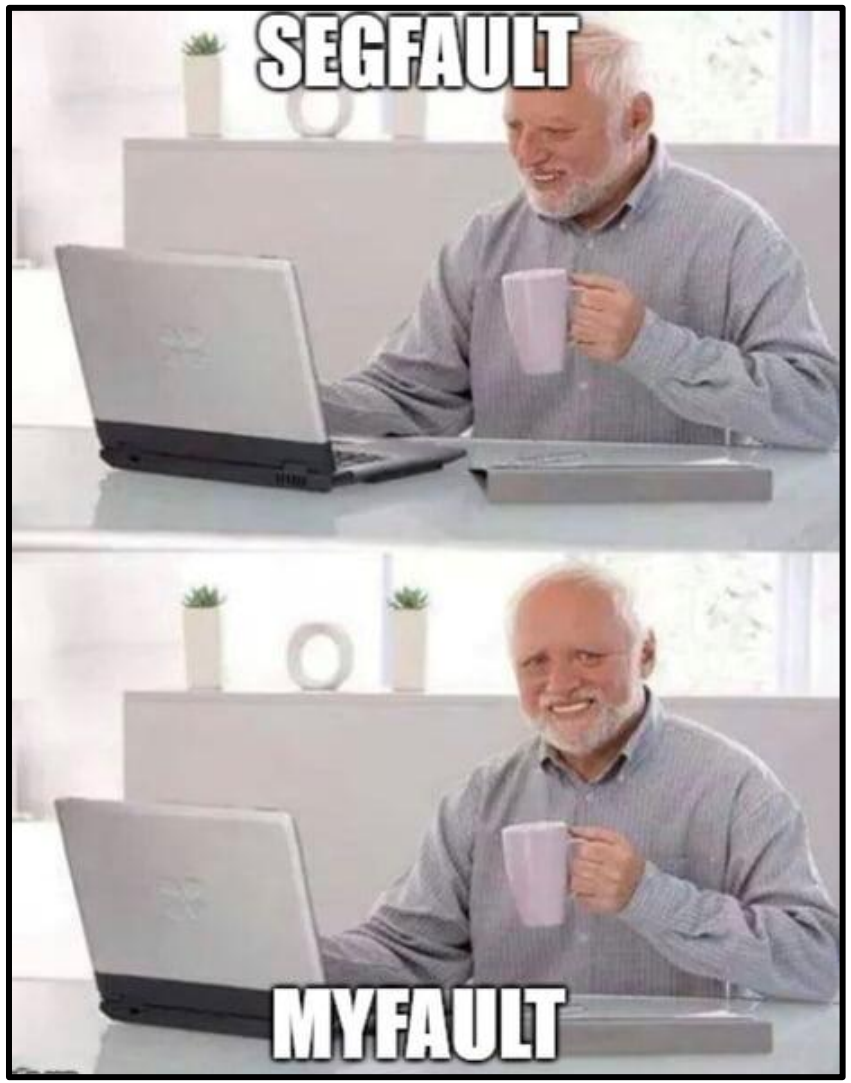
I/O

STL Data Structures

Makefiles and building

C/C++

- Both are efficient
- Both highly portable across platforms
  - Unix, windows, mac, android, etc
- Both require programmer-level memory management
- Programs can use both in the same program
  - C++ is a superset of C



C or C++?

C

- Simplicity
- Familiarity from previous course
- Easier syntax errors
- Compiler: gcc

C++

- Classes, generic types, bool, namespaces, and more
- Data structures
- Subtleties: references, Pointers, etc.
- Compiler: g++

Recommendation: Use both

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

int main(int argc, char** argv) {
  cout << "Hello World!\n";
}
```

```
g++ hello.cpp
```

C for OS-level system calls

- Reading binary files
- Formatted output

C++ for

- Classes
- New/delete
- Built-in string type, bool
- Data Structures
- Opportunity to learn it

Put all code in .cpp and use g++ to compile.

The stack

Execution stack – keeps track of active functions

Stack frame – created by each function
- contains parameters, local variables
- topmost frame is currently executing function
- frame is pushed to stack when function is called
- frame is popped off stack when it returns

We will use **stack diagrams** to visualize program flow
- https://pythontutor.com/c.html for checking your work

Example: What is the output of this program?

```
#include <iostream>  // cout, cin
#include <string>    // string class
#include <cmath>     // sqrt, cos, acos, log, etc

using namespace std;
int main(int argc, char** argv)
{
  int a = 0;
  string greeting = "hi";
  greeting += "?";
  float test = 3.14f;
  bool isCold = false;
  cout << greeting << endl;

  float fraction = 1/3;
 cout << fraction << endl;
}
```

The heap

Statically allocated objects live on the stack

- Automatically cleaned up when the function completes

Dynamically allocated objects live on the heap

- The user MUST clean up dynamically created objects themselves

Exercise: What is the output of this program?

```
#include <iostream>  // cout, cin
#include <string>    // string class
using namespace std;

struct Point {
  float x;  float y;
};

int main(int argc, char** argv) {
  Point* p = new Point();
  p->x = 1.0f; p->y = 2.0f;
  cout << p->x << " " << p->y << endl;
  delete p;

  p = (Point*) malloc(sizeof(Point));
  p->x = -2.0f;  p->y = -4.0f;
  printf("%f %f\n", p->x, p->y);
  free(p);
}
```

Helpful tools: valgrind and gdb

Use **valgrind** to test for memory leaks and corruption.

Use GDB to debug your program.

Exercise: what is the output of this program?

```
// Box.h
class Box {
 public:
   Box(float s) {
       mySize = s;
   }
  float getSize() {
      return mySize;
  }
 protected:
   float mySize = 1.0f;
};
```

```
int main(int argc, char** argv) {
   Box box1;
   float v1 = box1.value();

   Box* box2 = new Box();
   box2->value();
   delete box2;
}
```

Gotchas: Working with memory

NEVER mix malloc/free with new/delete

NEVER mix the array and object forms of new/delete

NEVER use malloc/free with classes

NEVER de-allocate objects on the ~~heap~~ Stack

Works or derps?

```
// Box.h
class Box {
 public:
   Box(float s) {
       mySize = s;
   }
  float getSize() {
      return mySize;
  }
 protected:
   float mySize = 1.0f;
};
```

```
int main(int argc, char** argv) {
   Box box1;
   float v1 = box1.value();

   Box* box2 = new Box();
   box2->value();
   free(box2);
}
```

Works or derps?

```
// Box.h
class Box {
 public:
   Box(float s) {
       mySize = s;
   }
  float getSize() {
      return mySize;
  }
 protected:
   float mySize = 1.0f;
};
```

```
int main(int argc, char** argv) {
   Box box1;
   float v1 = box1.value();

   Box* box2 = new Box();
   box2->value();
   delete[] box2;
}
```

Works or derps?

```
// Box.h
class Box {
 public:
   Box(float s) {
       mySize = s;
   }
  float getSize() {
      return mySize;
  }
 protected:
   float mySize = 1.0f;
};
```

```
int main(int argc, char** argv) {
   Box box1;
   float v1 = box1.value();

   Box* box2 = new Box();
   box2->value();
   delete box1;
}
```

Works or Derps?

```
#include <iostream>

int main() {
  int a[4] = {-1,-2,-3,-4};

  int* b = new int[5];
  for (int i = 0; i < 5; i++) {
    b[i] = i+1;
  }

  delete[] b;
  return 0;
}
```

Works or Derps?

```
#include <iostream>

int main() {
  int a[4] = {-1,-2,-3,-4};

  int* b = new int[5];
  for (int i = 0; i < 5; i++) {
    b[i] = i+1;
  }

  delete b;
  return 0;
}
```

Works or Derps?

```
#include <iostream>

int main() {
  int a[4] = {-1,-2,-3,-4};

  int* b = new int[5];
  for (int i = 0; i < 5; i++) {
    b[i] = i+1;
  }

  free(b);
  return 0;
}
```

Works or Derps?

```
#include <iostream>

int main() {
  int a[4] = {-1,-2,-3,-4};

  int* b = malloc(sizeof(int) * 5);
  for (int i = 0; i < 5; i++) {
    b[i] = i+1;
  }

  delete[] b;
  return 0;
}
```

Gotchas: parameter passing

- C supports *pass by value* and *pass by pointer*

- C++ adds *pass by reference*

BEST PRACTICE: Always pass by const reference
UNLESS you have a reason not to

Example: Parameter passing

```
void example(Box box) {
    box.size = 20;
}
```

```
void example(const Box& box) {
    int v = box.size;
}
```

```
void example(Box* box) {
    box->size = 20;
}
```

```
void example(Box& box) {
    box.size = 20;
}
```

Works or Derps?

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

void foo(string& text)
{
    text = “banana”;
}

int main(int argc, char** argv)
{
    string word = “apple”;
    foo(word);
}
```

Works or Derps?

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

void foo(const string& text)
{
    text = “banana”;
}

int main(int argc, char** argv)
{
    string word = “apple”;
    foo(word);
}
```

Gotchas: Return values

NEVER return a local or temporary variable!

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

string foo()
{
    string word = “apple”;
    return word; // creates a copy
}

int main(int argc, char** argv)
{
    string word = foo();
}
```

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

string& foo()   // Almost NEVER return reference
{
    string word = “apple”;
    return word; // returning an object that will be deleted
}

int main(int argc, char** argv)
{
    string word = foo();
}
```

Works or Derps?

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

string foo()
{
    string word = “apple”;
    return word;
}


int main(int argc, char** argv)
{
    string word = foo();
}
```

Works or Derps?

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

string& foo()
{
    string word = “apple”;
    return word;
}


int main(int argc, char** argv)
{
    string word = foo();
}
```

C++ Rules of Thumb

- Return by value
- Avoid unnecessary copying
- Pass parameters as const reference unless the function modifies the parameter
- Only use pointers when necessary. Assume that NULL values are possible whenever you use pointers.
- NEVER mix malloc/free with new/delete
- If you allocate memory within a class, delete the memory in the same class
- Only break the rules when you have a clearly defined reason to do so.

Console I/O

Three default streams

- stdin (scanf, fscanf)
- stdout (printf, fprintf)
- stderr (fprintf(stderr, <format string>, <data>))

stdin, **stdout**, **stderr** are tied to files with descriptors 0, 1, and 2 respectively

At the command line we can redirect these streams to files

- ./a.out < infile.txt
- ./a.out > outfile.txt
- ./a.out &> allfile.txt (all output)
- ./a.out 2> errfile.txt (error output only)
- ./a.out 1> outfile.txt is the same as ./a.out outfile.txt

Standard I/O (C)

```
int a;
 char word[32];
 printf("Enter an integer: ");
 scanf(" %d", &a);

 printf("Enter a string: ");
 scanf(" %s", word);

 printf("\nYou entered: %d %s\n",
              a, word);
 fprintf(stderr, "Test printing an error\n");
```

```
int a;
 char word[32];
 fprintf(stdout, "Enter an integer: ");
 fscanf(stdin, " %d", &a);

 fprintf(stdout, "Enter a string: ");
 fscanf(stdin, " %s", word);

 fprintf(stdout, "\nYou entered: %d
%s\n",
              a, word);
 fprintf(stderr, "Test printing an error\n");
```

printf(...) is the same as fprintf(stdout, ...)

When we redirect output/input, they are fed into/out of these commands
(demo)

Standard I/O (C++)

```
int a;
std::string word;
std::cout << "Enter an integer: ";
std::cin >> a;

std::cout << "Enter a string: ";
std::cin >> word;

std::cout << std::endl << "You entered: " << a << " " << word << std::endl;
std::cerr << "Test printing an error" << std::endl;
```

cout is the same as fprintf(stdout, ...), etc

This program works the same as the previous two

Reading Text Files (C)

```
const char* filename = "grades.txt";
FILE* fp = fopen(filename, "r");
if (!fp) // fp is NULL on error
{
  printf("Cannot load file: %s\n", filename);
  return 1;
}
int num = 0;
float sum = 0;
char line[1024];
while (fgets(line, 1024, fp))
{
  int grade;
  sscanf(line, " %d", &grade);
  sum += grade;
  num++;
}
printf("The average is %.2f\n", sum/num);
fclose(fp);
```

```
99
79
51
92
56
89
63
```

Reading Text Files (C++)

```
ifstream file(filename);
if (!file) // true if the file is valid
{
  cout << “Error: " << filename << endl;
  return 1;
}
int num = 0;
float sum = 0;
int grade = 0;
string line;
while (getline(file, line))
{
  num++;
  stringstream ss(line);
  ss >> grade;
  sum += grade;
}
cout << "The average is " << fixed << showpoint << setprecision(2) <<
  (sum/num) << endl;
file.close();
```

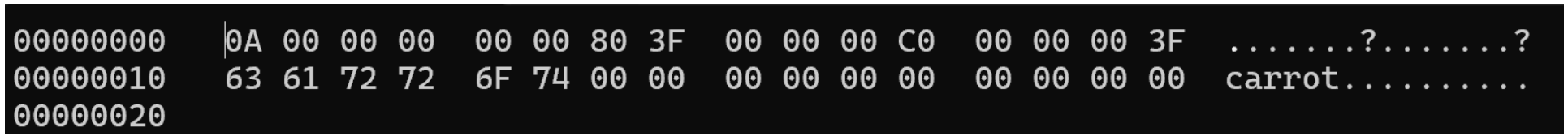
```
99
79
51
92
56
89
63
```

Question: How is redirected input/output different from reading/writing a file?

Writing Binary Files

```
#include <cstdio>
struct Data {
  int size;
  float p[3];
  char buff[16];
};

int main()
{
  struct Data data = {10, 1.0, -2.0, 0.5, "carrot"};
  FILE* fp = fopen("data.bin", "wb");
  fwrite(&data, sizeof(struct Data), 1, fp);
  fclose(fp);
}
```

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

struct Data {
  int size;
  float p[3];
  char buff[16];
};

int main()
{
  Data data = {10, 1.0, -2.0, 0.5, "carrot"};
  ofstream file = ofstream("data.bin", ios::binary);
  file.write((char *) &data, sizeof(Data));
  file.close();
}
```



Reading Binary Files

```
struct Data data;
 FILE* fp = fopen("data.bin", "rb");
 fread(&data, sizeof(struct Data), 1, fp);
 fclose(fp);
```

```
Data data;
 ifstream file = ifstream("data.bin", ios::binary);
 file.read((char *) &data, sizeof(Data));
 file.close();
```

```
00000000   0A 00 00 00  00 00 80 3F  00 00 00 C0  00 00 00 3F  .......?.......?
00000010   63 61 72 72  6F 74 00 00  00 00 00 00  00 00 00 00  carrot..........
00000020
```

Reading Binary Files II

You can move the file pointer to any place in the file.

Useful for reading data structures from a file

```
FILE* fp = fopen("data.bin", "rb");
fread(&data, sizeof(struct Data), 1, fp);

float x[3];
fseek(fp, sizeof(int), SEEK_SET); // Go 4 bytes from the beginning of the file
fread(&x, sizeof(float), 3, fp);

fseek(fp, 0, SEEK_END); // go to end
size_t filesize = ftell(fp); // get fp offset in bytes
fclose(fp);
```

```
00000000   0A 00 00 00  00 00 80 3F  00 00 00 C0  00 00 00 3F  .......?.......?
00000010   63 61 72 72  6F 74 00 00  00 00 00 00  00 00 00 00  carrot..........
00000020
```

Pipes

Idea: the output of one program is fed into another program as input.

```
alinen@sutekh:~/cs355/os-devel/lectures$ find . -name "*.c"
./week01/fileIO-C-Text.c
./week01/stdio0.c
./week01/stdio1.c
alinen@sutekh:~/cs355/os-devel/lectures$ find . -name "*.c" | grep st
./week01/stdio0.c
./week01/stdio1.c
alinen@sutekh:~/cs355/os-devel/lectures$
```

C++ Standard Library Data Structures

std::vector – dynamic array

std::list – linked list

std::map - dictionary

std::unordered_map – hash map

std::string: what is the output?

```
string phrase = "the quick, brown dog";

if (phrase.find("quick") != string::npos) {
  cout << "Found quick in phrase!\n";
}

cout << "The string length is " << phrase.size() << std::endl;

string newphrase = "";
for (unsigned int i = 0; i < phrase.size(); i++) {
   if (phrase[i] == 'i') newphrase += "1";
   else if (phrase[i] == 'o') newphrase += "0";
   else if (phrase[i] == 'e') newphrase += "3";
   else newphrase += phrase[i];
}

cout << "newphrase: " << newphrase << endl;

string filename = "hello.txt";
string newfilename = filename.substr(0, filename.size()-4) + ".bvh";
cout << "newfilename: " << newfilename << endl;
```

std::vector: what is the output?

```
vector<float> values = {1.0, -2.0, 3.0};
values.push_back(-4.0);

for (unsigned int i = 0; i < values.size(); i++) {
   cout << values[i] << endl;
}

values.clear();
values = vector<float>(10);
for (unsigned int i = 0; i < 10; i++) {
  values[i] = i;
}

values[1] *= 10.0;
for (float v : values) {
  cout << v << endl;
}
```

See: vector demo

std::list

```
  list<int> values = {1, -2, 3};
  values.push_back(-4);
  for (auto it = values.begin(); it != values.end(); it++) {
    cout << *it << endl;
  }
  values.clear();
  values.push_front(25);
  values.push_back(13);

  // Insert before 16 by searching
  auto it = find(values.begin(), values.end(), 13);
  if (it != values.end())  values.insert(it, 42);

  cout << "l = { ";
  for (int n : values)
    cout << n << ", ";
  cout << "};\n";
}
```

See: list demo

std::map

```
map<string,int> names2age;
names2age["giles"] = 54;
names2age["buffy"] = 18;
names2age["joyce"] = 38;

cout << "Number of items: " << names2age.size() << endl;

for (auto it = names2age.begin(); it != names2age.end(); ++it) {
  cout << it->first << ", " << it->second << endl;
}

names2age.clear();
names2age = { {"giles", 54}, {"buffy", 18}, {"drusilla", 176} };

for (auto [key, value] : names2age) {
  cout << key << ", " << value << endl;
}
```

See: map demo

Makefiles

We can build using either g++/gcc or make. What is the difference?

What are the advantages makefiles?

Example Makefile (gmake)

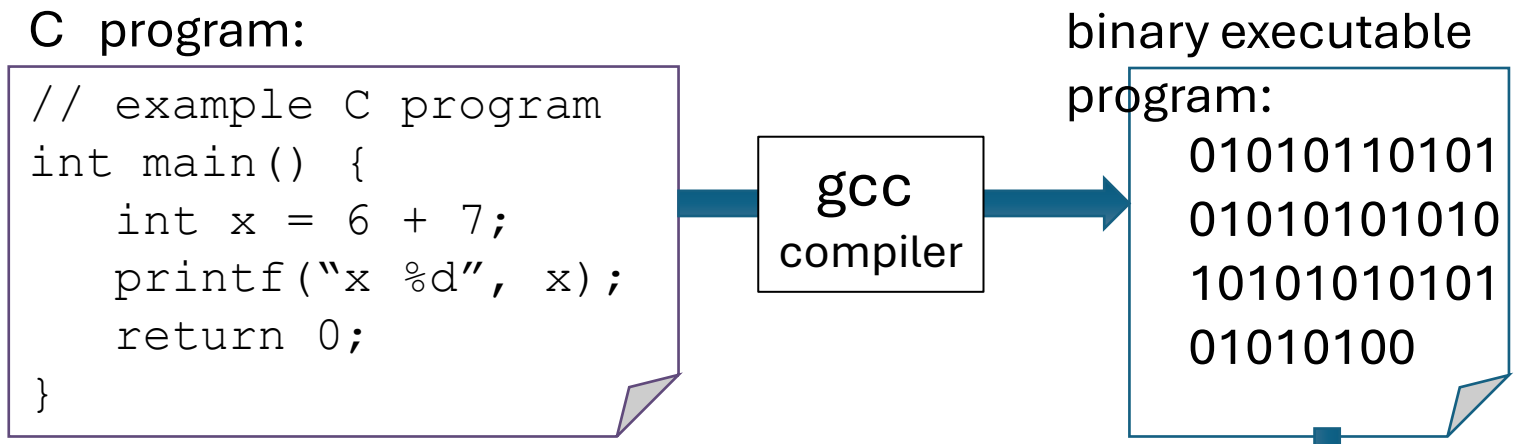
```
CC=g++
CSOURCES=$(wildcard *.c)
CPPSOURCES=$(wildcard *.cpp)
FILES := $(subst .c,,$(CSOURCES)) $(subst .cpp,,$(CPPSOURCES))
FLAGS=-g -Wall -Wvla -Werror -Wno-unused-variable -Wno-unused-but-set-variable

all: $(FILES)

% :: %.c
  $(CC) $(FLAGS) $< -o $@

% :: %.cpp
  $(CC) $(FLAGS) $< -o $@

clean:
  rm -rf $(FILES)

print:
  $(info $(CPPSOURCES) $(CSOURCES))
```

This makefile can build any single file C/C++ program

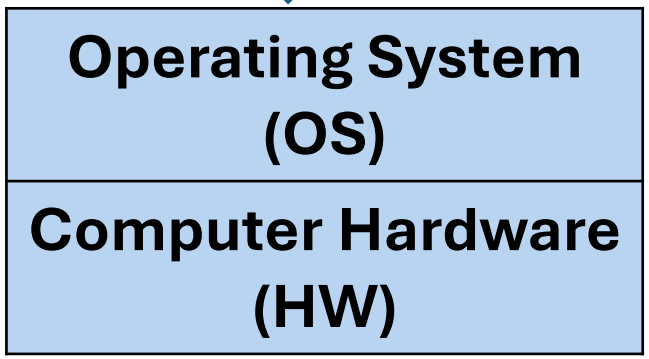
Building and Running a C program

Compiling a C program translates it to binary (0's and 1's)

The binary file is an **executable**, meaning "we can run it"



2. With OS's help, HW circuits runs binary executable



Building larger C/C++ programs

Header files contain definitions for classes, variables, functions (.h, .hpp)

Source files contain implementation (.c, .cpp)

- Header files can also contain implementation, but it is typically kept separate

Very large programs are built in steps

- Compile all source files into binary files (.o, .a)
  - → Checks for syntax errors and undefined symbols (e.g. missing definitions)
- Link all binaries together into an executable
  - → Checks that all definitions for classes and functions have implementations

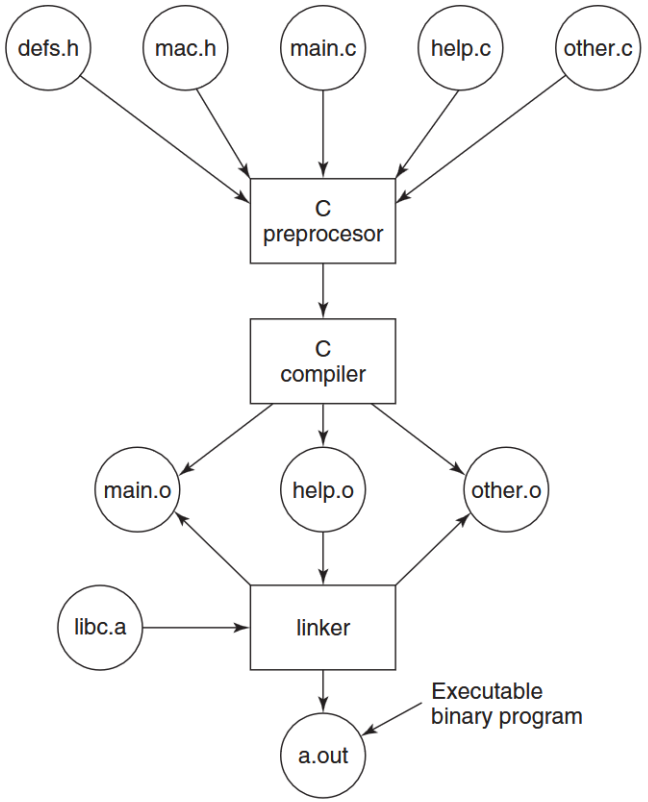


Figure 1-30. The process of compiling C and header files to make an executable.

Libraries and Linking

What is the difference between an executable and library?

Give some examples of libraries.