From C to C++

Overview and Examples

References

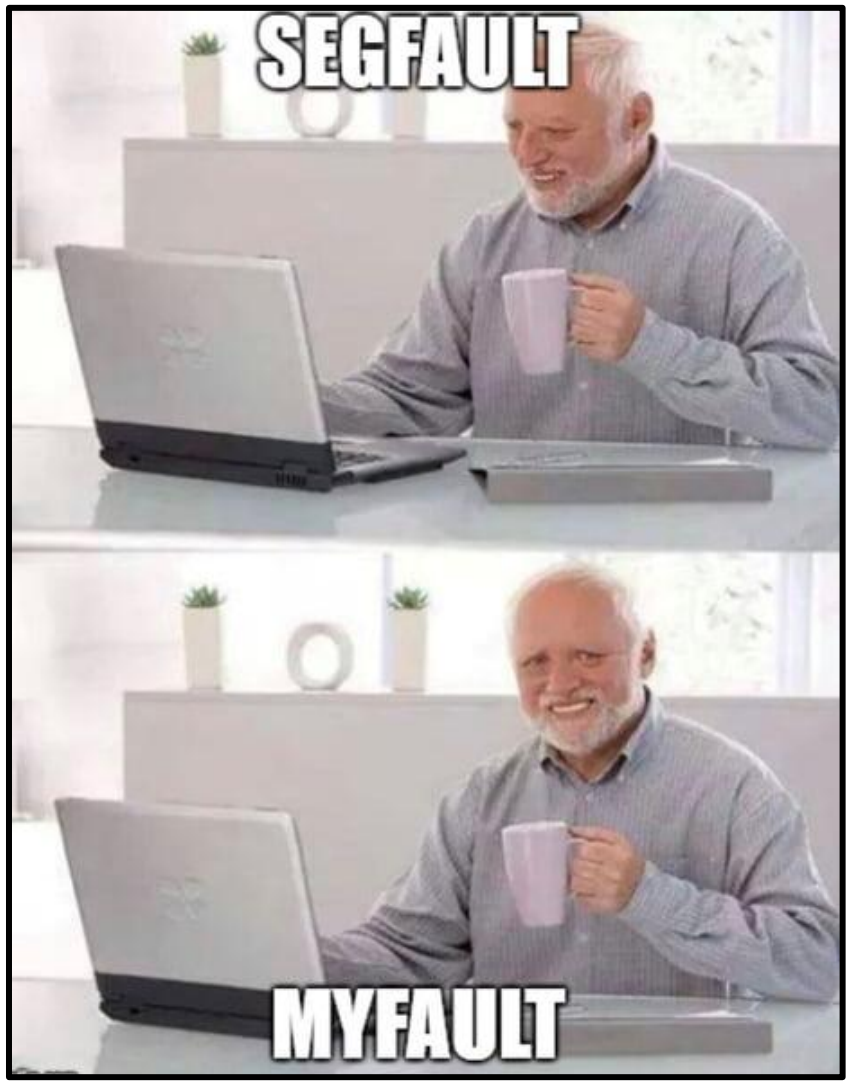
Classes that manage memory

STL Data Structures

Generic programming (Templates)

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
- Linux system calls are C
- 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.

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;
}
```

Aside: namespaces

Method for encapsulating APIs to avoid name conflicts

Example: Multiple APIs define a vector object in their own namespace. Users of the API can use both vector types with conflicts

Example: Built-in high-level C++ functionality is in the STD namespace

```
// Syntax with using statement
#include <iostream>  // cout, cin
using namespace std;
int main(int argc, char** argv)
{
  cout << “Hello!” << endl;
  return 0;
}
```

```
// Syntax without using statement
#include <iostream>  // cout, cin

int main(int argc, char** argv)
{
  std::cout << “Hello!” << std::endl;
  return 0;
}
```

NEVER put a `using` statement in a header file.

C++ Defining classes

```
class Box // Java
{
  protected float mySize = 1.0f;
  public Box(float s)
  {
     mySize = s;
  }

  public float getSize()
  {
     return mySize;
  }
}
```

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

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

C uses malloc/free

C++ adds free/delete/delete[]

Example: Compare these two programs

```
#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;
}
```

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

Example: Compare these two programs

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

int main(int argc, char** argv) {
  int* array = new int[5];
  for (int i = 0; i < 5; i++) {
    array[i] = 10;
  }
  delete[] array;
}
```

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

int main(int argc, char** argv) {
  int* array = (int*) malloc(sizeof(int) * 5);
  for (int i = 0; i < 5; i++) {
    array[i] = 10;
  }
  free(array);
}
```

Example: 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;
}
```

Unlike JAVA, C++ allows you to create objects on either the stack or heap

Example: Classes and header files

In C, we use header files to split declarations from implementations. C++ is the same.

```
// foo.h
#ifndef ex_H_
#define ex_H_

extern int myfoo();

#endif ex_H_
```

```
// foo.c
#include “ex.h”

int myfoo() {
   // implementation
}
```

Function definitions split between header and source files

```
// myclass.h
#ifndef myclass_H_
#define myclass_H_

class A {
public:
   A();
   int foo() const;
}

#endif myclass_H_
```

```
// myclass.cpp
#include “myclass.h”

A::A() {
   // implementation
}

int foo() const {
  // implementation
}
```

Class definitions split between header and source files

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 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;
}
```

```
#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;
}
```

```
#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;
}
```

References

C++ adds an implicit pointer type called a **reference**

References alias a variable: you can make changes to it without copying it. Syntax:

```
const Point& point = getPosition(); // const references cannot be changed
Point& point = vertices[2];         // changing point modifies the original data
```

Unless it’s a basic type, always pass parameters by reference unless you have a reason not to.

C++ 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);
}
```

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

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

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

C++ Return values

Same as C: 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();
}
```

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

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


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

Details: const in C++

const helps us avoid mistakes by telling the compiler when data should not be modified

```
const int a = 5; // constant value
class A {
   ...
   int foo(const A& other);
   int bar() const;
};
```

const parameters allow us to set values anonymously

```
A a;
a.foo(A()); // pass default a object to foo
```

C++ Rules of Thumb

- Return by value
- Avoid unnecessary copying
- Pass parameters as const reference unless the function modifies the parameter
- Define member functions as const whenever possible
- Only use pointers when you want variables that can have NULL values. Always treat NULL as a valid pointer value.
- 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.

Recall: 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

The C functions are faster

To get a C string from a C++ string, call `cstring.c_str()`

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
```

Writing Binary Files

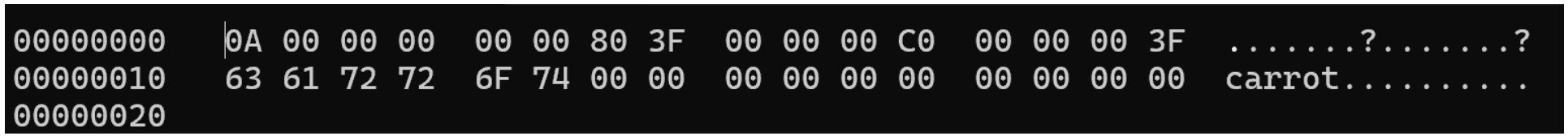
```
#include <cstdio>
struct Data {
  int size;
  float p[3];
  char buff[16];
};

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
```

C++ Standard Library Data Structures

std::string – resizable string type

std::vector – dynamic array, like ArrayList in JAVA

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

C++ Class Design

OOP: Objects encapsulate functions (e.g. API) and data together

- Objects “know stuff” and can “do stuff”
- Supports modularity when objects are self-contained with well-defined interfaces

Classes/objects are how we can extend the language with our own data types. Data types can require

- Their own operators: stream (for printing), [], +, ==, etc.
- Copy, assignment, and destruction implementation

Example: MyString

Let’s define a class to make a user-friendly string class

Features:

- MyString should manage an underlying array of characters, e.g. a C string
  - MyString should not leak memory!
- MyString a = b should create a new string object, initialized with the contents of b
- MyString a should create an empty string
- MyString a = “hello” should initialize a new string with the given constant
- a = b should copy the contents of b into a
- a[index] should return the index’th character of a
- a == b should return true iif the contents of a lexographically match the contents of b, e.g. a == b when strcmp(a,b) == 0
- a.c_str() should return a char* pointer to the underlying character buffer
- a.size() should return the length of the string
- cout << a should print the string

Constructors, destructors, and assignment

C++ defines default implementations for creating, deleting, and assigning objects but when our class must manage memory, we cannot rely on them

```
MyString(); // Default ctor. Called when no args given
MyString(const MyString& other); // Copy ctor. Called on assignment to new variable
MyString& operator=(const MyString& other); // Assignment operator
virtual ~MyString(); // Destructor. Called when an object is deleted (stack or heap!)
```

Sketch implementations for these functions

Constructors, destructors, and assignment: Implementation sketches

Exercise: Who is called?

```
int main() {
  MyString a;                  // 1
  MyString b = "hello";  // 2
  MyString c = b;            // 3
  a = b;                          // 4

  MyString* d = new MyString();  // 5
  *d = "bye";                              // 6
  delete d;                                 // 7

  MyString e = MyString(“eee”);  // 8
  MyString f(“eff”);                    // 9
  return 0;
}   // 10
```

Stream operators

C++ uses global functions to print to streams

A stream is a sequence of characters, such as from stdin or stdout

e.g. `cout << mystring < endl` prints mystring to stdout

Implementing the output stream operator in MyString

1. Define the global stream function
2. Make the stream function a “friend” so that we can print out non-public instance variables

```
class MyString {
 ...
 friend std::ostream& operator<<(std::ostream& os, const MyString& str);
 ...
};

std::ostream& operator<<(std::ostream& os, const MyString& str) {
  os << str.m_buf;
  return os;
}
```

Example: [] operator

Square bracket indexing

```
char& operator[](int idx) {  // A
    return m_buf[idx];
}

const char& operator[](int idx) const {  // B
   return m_buf[idx];
}
```

```
// Who is called?
const MyString f("fff");
 for (int i = 0; i < f.size(); i++) {
   printf("%c\n", f[i]);   // 1
 }

for (int i = 0; i < a.size(); i++) {
   printf("%c\n", a[i]);  // 2
 }

 a[0] = ‘y’;  // 3
```

Exercise: Sketch equality operator with ==

Goal: Write code that supports the following syntax

```
MyString a(“aaa”);
MyString b(“bbb”);
if (a == b) {
    // stuff
}
```

Generic programming

C++ templates allow you to write functions and objects that work with any type

- Example: A linked list that works with any data
- Example: A function that can compute the max of any type that supports <

C++ templates tend to be fast because the compiler converts template code to inline assembly instructions

Example: function

What code is generated?

```
template<typename Type>
Type myMax(Type a, Type b) {
   return a > b ? a : b;
}
```

```
int main() {
  float f = myMax(1.0, 2.0);
  char c = myMax('a', 'A');
  return 0;
}
```

Example: Linked list

Exercise: Implement print, insert_first, and the destructor

```
template<typename T>
struct Node {
  T val;
  struct Node<T>* next;
};

template<typename T>
class LinkedList {
public:
  LinkedList() {
    m_head = nullptr;
    m_size = 0;
  }

  ....// etc


private:
  unsigned int m_size;
  Node<T>* m_head;
};
```

```
struct Movie {
  int duration;
  char title[128];
};

std::ostream& operator<<(std::ostream& of, const Movie& movie) {
  of << movie.title << "(" << movie.duration << ")";
  return of;
}

int main() {
  LinkedList<Movie> movies;
  movies.insert_front(Movie{190, "Attack of the Phones"});
  movies.insert_front(Movie{90, "Dreams of Quiet"});
  movies.insert_front(Movie{80, "Wild Fur"});

  movies.print();
}
```

Exercise: LinkedList with templates

C++ Next Steps

C++ is “confederacy of languages”, meaning it supports many features not covered here

Many C++ features aim to make memory management easier/safer and handling template types easier, e.g.

- Example: smart pointers make it easier to manage pointers that are passed around in a codebase (e.g. unique_pointer, shared_pointer)
- Example: *auto* can be used in lieu of a data type
- Example: Loops support foreach syntax

C++/C like languages such as Rust focus on safety and concurrency