# Classes

## Intermediate Computer Systems Programming, Fall 2025

**Instructor:** Travis McGaha

**TAs:** Theodor Bulacovschi Ash Fujiyama

# Poll Everywhere

**pollev.com/tqm**

- How is the workload going so far? Is it ok?

# Administrivia

- First Assignment (HW00 cowsay)
  - "Due" last week
  - Extended to be due Tuesday the 9th (course selection period ends)
  - Mostly a C refresher
  - Don't put it off, HW01 is posted and you have to work on that too ☺

- Pre semester Survey
  - Anonymous
  - Due Friday the 12th

- HW01 Posted tonight
  - Due Yesterday the 9th
  - More rigorous C refresher ☺

# Administrivia

- First Check-in (Check-in00)
  - “Due” before lecture today
  - Extended to be due before lecture on Wednesday
  - It tells you when you are correct! Unlimited Attempts!

- Next HW (HW02)
  - Out after class on Wednesday
  - Due on the 16th ☺
  - C++ Classes!

# Lecture Outline

- **Integer types in C++**
- Modern C++: std::string!
- Constructing, Destructing & Methods!
- Copying
- Struct vs Class and “regular” types

# Integer Types in C++

- Anyone remember how big an int is?
- Is a char signed or unsigned?
- How big is an unsigned long?
  What about an unsigned long long?
- For many of the base primitive types C++ leaves things ambiguous. So for the most part we will prefer other integer types
  - Int and char are still fine in some cases, just be sure that you know what you are doing!

# Fixed Width Integers

- For when you want to be specific about memory size, you should use fixed with integers specified in `<cstdint>`
  - Some people/workplaces will always use these instead of "plain types" (especially embedded related projects)
- E.g.
  - `int8_t` signed 8 bit integer
  - `uint8_t` unsigned 8 bit integer
  - `int16_t`
  - `uint16_t`
  - `int32_t`
  - `uint32_t`
  - `int64_t`
  - `uint64_t`

# size_t

- Defined in `<cstddef>` and other headers
- An unsigned integer size big enough to hold the size (in bytes) of any possible object or array in memory.

- Often used for indexes, loop counters, etc.
- The C++ standard library uses it as the container specified size_type.
  - Other libraries like may use a different size_type for their containers (QT for example just uses int)

- ssize_t is a signed type (not part of C++ officially but is in Linux C)
  - Same size as size_t but it is a signed type.
  - Usually used only when a function could possibly return error, so -1 is returned.

# Don’t mix and match

- Bad things happen, especially when you mix signed and unsigned types

- Some workplaces avoid using unsigned types altogether
  - Some languages (e.g. Java) don’t have unsigned types to avoid these issues.

- Preferred strategy:
  - stick with one consistent type. Already dealing with `size_t`’s? then stick to `size_t`
  - Using unsigned types? Avoid subtraction (e.g. `i < len - 1` becomes `i + 1 < len`)

- If need to have multiple types, keep it minimal. static_cast can be used to be more explicit about the types you are using.

# static_cast

Any well-defined conversion

- static_cast can convert:
  - casting void* to T*
  - Non-pointer conversion
    - *e.g.* float to int

```
void foo() {
  int b = 3;
  float c;

  c = static_cast<float>(b);
}
```

- If you are doing a cast not related to object inheritance, it will most likely be this one.

- There are other C++ casts, but this is the only one you need for now. Don't use C style casts anymore.

# Lecture Outline

- Integer types in C++
- **Modern C++: std::string!**
- Constructing, Destructing & Methods!
- Copying
- Struct vs Class and "regular" types

# use_string.cpp example walkthrough

```
#include <string>

int main() {
  // construct a string
  string hi{"hello there"};

  // this also works:
  string hi2 = "hello there";

  cout << hi << endl; // print the string

  // both of these work to get the length
  size_t len = hi.size();
  len = hi.length();


  // get/set the first character
  char c = hi.at(1);
  hi.at(5) = '_';
  // char oops  = hi.at(3000); // throws out_of_range exception
  // char oops2 = hi[3000];    // Unchecked access: does not throw, do not use
```

# use_string.cpp example walkthrough

```
int main() {
  // construct a string
  string hi{"hello there"};

  // two ways to iterate over each character
  for (char c : hi) {
    cout << c << endl;
  }

  for (size_t i = 0; i < hi.size(); ++i) {
    cout << hi.at(i) << endl;
  }

  // find and substr
  size_t index = hi.find("t");
  if (index == string::npos) {
    // not found
    cerr << "\"t\" not found" << endl;
  } else {
    // first param is starting offset.
    // second param is the length NOT the ending index.
    string sub = hi.substr(1, 4);
  }
```

# Objects!

- In C++ we have objects!
- See how this is easier to use than SimpleString?
- Notice that it handled the memory for us! Both allocating and deallocating ☺

- We will show you how to make classes in C++, but first we will show you how to make better structs

# Lecture Outline

- Integer types in C++
- Modern C++: std::string!
- **Constructing, Destructing & Methods!**
- Copying
- Struct vs Class and “regular” types

# Demo: Adding Constructors

- Lets build off of SimpleString from HW00
- Previously we had:

```
struct SimpleString {
  char* data;
  size_t len;
};

int main() {
  SimpleString str = SimpleString_From("Hello");
}
```

- Lets add a constructor!

# Demo: Adding Constructors

- Lets add a constructor!
- In .hpp:

```
struct SimpleString {
  char* data;
  size_t len;


  // declare a constructor
  SimpleString(char* cstr);
};
```

Structs can have both:
- data members
- Member functions

SimpleString::
declares the function as a member of the SimpleString struct

- In .cpp:

Constructor function signature

```
SimpleString::SimpleString(char* cstr) {



}
```

# Demo: Adding Constructors

- Lets add a constructor!
- In .hpp:

```
struct SimpleString {
  char* data;
  size_t len;


  // declare a constructor
  SimpleString(char* cstr);
};
```

Initialize data members in an "initializer list"
should be in the same order they are declared in the struct
these are run before body of constructor
`member_name(expression)`

- In .cpp:

```
SimpleString::SimpleString(char* cstr) : data(new char[strlen(cstr) + 1]), len(strlen(cstr)) {
  for (size_t i = 0; i <= len; ++i) {
    data[i] = cstr[i];
  }
}
```

# Destructors & RAII

- When are we done with the variables `str` and `howdy`?

```
void print_str(string input) {
  cout << input << endl;
}


void func(int input) {
  if (input % 2 == 0) {
    string howdy{"howdy: "};
    cout << howdy;
  }
  cout << input << endl;
}
```

When they fall out of scope (hit the closing brace that ends the scope they are declared in)

# Destructors

- C++ has the notion of a destructor (dtor)
  - **Invoked automatically when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)**
  - Place to put your cleanup code – free any dynamic storage or other resources owned by the object
  - Standard C++ idiom for managing dynamic resources
    - Slogan: "*Resource Acquisition Is Initialization*" (RAII)
    - Part of the object design (e.g. object invariant) is to hold ownership of some resource (char* pointing to the heap in our case). That resource is acquired at construction and held until destruction.

tilde

No parameters

```
MyObj::~MyObj() {    // destructor
  // do any cleanup needed when a "MyObj" object goes away
}
```

When a destructor is invoked:
1. run destructor body
2. Call destructor of any data members

# Demo: Adding Destructors

- In .hpp:

```
struct SimpleString {
  char* data;
  size_t len;

  // declare a constructor
  SimpleString(char* cstr);

  // decare a destructor
  ~SimpleString();
};
```

- In .cpp:

```
SimpleString::~SimpleString() {
  delete[] data;
}
```

# Destructor In Practice

- Destructor calls are inserted for us automatically whenever a function falls out of scope:

```
int main() {
  SimpleString str("Hello!");

  // compiler implicitly adds call to:
  // str.~SimpleString()
}
```

# Demo: Adding Methods

- Adding a normal member function
- In .hpp:

```
struct SimpleString {
  char* data;
  size_t len;

  SimpleString(char* cstr);

  ~SimpleString();

  // member function (method)
  char& at(size_t index);
};
```

Like java, we can use `this` to refer to the "object" we are calling the function on.

- In .cpp:

```
char& SimpleString::at(size_t index) {
  return this->data[index];
}
```

Usually it is optional we could also do `return data[index];`

If it helps, you can think of every function as having a secret parameter (`T * const this`) that points to the "object" we called the function on.

Just a function.
All we have to add is SimpleString:: to indicate it is a member of the SimpleString struct

## Ed Discussion

- Finish writing the Vec struct. A Vec (short for vector) is a dynamically resizable array (like ArrayList in Java).

- Vectors start with allocating an array and put the elements of their “list” in that array. When there isn’t enough space in the array, it allocates a new array and copies elements over.

# Lecture Outline

- Integer types in C++
- Modern C++: std::string!
- Constructing, Destructing & Methods!
- **Copying**
- Struct vs Class and "regular" types

**Raise Your Hands**

- What do you think happens in this case?

  Yes, this code compiles

```
void ALL_CAPS(string str) {
  for (size_t i = 0; i < str.length(); ++i) {
    str.at(i) = toupper(str.at(i));
  }
}


int main() {
  string mf = "doom";
  ALL_CAPS(mf);


  cout << mf << endl;
}
```

## Raise Your Hands

- What do you think happens in this case?
  Yes, this code compiles

```
void ALL_CAPS(string str) {
  for (size_t i = 0; i < str.length(); ++i) {
    str.at(i) = toupper(str.at(i));
  }
}


int main() {
  string mf = "doom";
  ALL_CAPS(mf);


  cout << mf << endl;
}
```

- It prints “doom”
- C++ objects try to mimic “pass by value” like primtives do.
- The str in ALL_CAPS is an intendent copy of mf
- How does this work?

# Copy Constructors

- C++ has the notion of a copy constructor, which constructs a new struct that is an independent copy of an already existing struct:

- In .hpp

```
struct SimpleString {
  char* data;
  size_t len;

  // declare copy constructor (cctor)
  SimpleString(const SimpleString& other);

};
```

- In .cpp:

```
SimpleString::SimpleString(const SimpleString& other) : data(new char[other.len + 1]), len(other.len)  {
  for (size_t i = 0; i <= len; ++i) {
    this->data[i] = other.data[i];
  }
}
```

# When Do Copies Happen?

- The copy constructor is invoked if:
  - You *initialize* an object from another object of the same type:

```
Point x;        // default ctor
Point y(x);     // copy ctor
Point z = y;    // copy ctor
```

  - You pass a non-reference object as a value parameter to a function:

```
void foo(Point x) { ... }

Point y;        // default ctor
foo(y);         // copy ctor
```

  - You return a non-reference object value from a function:

```
Point foo() {
  Point y;      // default ctor
  return y;     // copy ctor
}
```

# Why Copying is an issue: Depending on the type

- Copying means we have to make an independent copy of a variable.
  - **This requires iterating over the elements**
  - **This requires dynamic memory allocation.**

- The cost to make a copy varies on what type we are copying.
  - Do you think this "point" struct takes a lot to copy?
    - No: low time complexity and no memory allocation
  - What about a Hash Map?
    - Yes: need to reallocate all key/value pairs and iterate over the original hash map.

```
struct Point {
   ...
 private:
  int x_;
  int y_;
};
```

# Const reference most parameters

- To avoid creating copies, we pass most things in as a const references to the function.
  - References make it so we do not copy the object
  - Const makes it so that we do not accidentally modify the value if we do not need it to be modified

- For primitive types (other cheap types) passing a copy is fine and possibly faster.

```
void print_str(const string& input) {
  cout << input << endl;
}

void func(int input) {
  if (input % 2 == 0) {
    string howdy{"howdy: "};
    cout << howdy;
  }
  cout << input << endl;
}
```

# Assignment != Construction

- "=" is the assignment operator
  - Assigns values to an *existing, already constructed* object

```
Point w;               // default ctor
Point x(1, 2);         // two-ints-argument ctor
Point y(x);            // copy ctor
Point z = w;           // copy ctor
y = x;                 // assignment operator
```

Method operator=()

equivalent code:
`y.operator=(x);`

# Overloading the "=" Operator

- You can choose to define the "=" operator
  - But there are some rules you should follow:

Here checking against yourself is not important, but in other types it can matter

```
Point& Point::operator=(const Point& rhs) {
  if (this != &rhs) {   // (1) always check against this
    x_ = rhs.x_;
    y_ = rhs.y_;
  }
  return *this;         // (2) always return *this from op=
}

Point a;          // default constructor
a = b = c;        // works because = return *this
```

More important when data members are Dynamic memory

Should be a reference to *this to allow chaining

Explicit equivalent:
`a.operator=(b.operator=(c));`

## Ed Discussion

- Lets say we don't check for assigning to self in operator=
- What line has a crash/memory error?

```
SimpleString& SimpleString::operator=(const SimpleString& rhs) {
  if (this != &rhs) {
    delete[] this->data;

    this->data = new char[rhs.len + 1];
    this->len = rhs.len;

    for (size_t i = 0; i <= len; ++i) {
      data[i] = other.data[i];
    }
  }
  return *this;
}
```

## Ed Discussion

- Add a copy constructor and an assignment operator to our Vec struct from the previous poll

## Ed Discussion

- How many strings are made?

- If you have time:
  - How many times is a string destructor run?
  - What does this print?

```
string reverse(string input) {
  string res{};
  res = input;
  for (size_t i = 0; i < input.length(); ++i) {
    res.at(i) = input.at((input.length - 1) - i);
  }
  return res;
}


int main() {
  string nums {"3192"};
  string result = prefix_sum(nums);
  for (char c : pre_sum) {
    cout << i << endl;
  }
}
```

# Lecture Outline

- Integer types in C++
- Modern C++: std::string!
- Constructing, Destructing & Methods!
- Copying
- **Struct vs Class and "regular" types**

# Access Modifiers

- Typically we want our data members to be private, and we can make this happen with access modifiers:

Note: access modifier applies to all following members until a new modifier is specified

```
struct SimpleString {
  char* data;
  size_t len;

  // declare a constructor
  SimpleString(char* cstr);


  // decare a destructor
  ~SimpleString();

  // member function (method)
  char& at(size_t index);
};
```

```
struct SimpleString {
 public:
  // declare a constructor
  SimpleString(char* cstr);

  // decare a destructor
  ~SimpleString();

  // member function (method)
  char& at(size_t index);

 private:
  char* data;
  size_t len;
};
```

# Class vs Struct (Functionality)

- A class is the same thing as a struct. Only difference is if you don't specify an access modifier, class assumes private, structs assume public. THAT'S IT

```
struct SimpleString {
 public:
  // declare a constructor
  SimpleString(char* cstr);

  // decare a destructor
  ~SimpleString();

  // member function (method)
  char& at(size_t index);

 private:
  char* data;
  size_t len;
};
```

```
class SimpleString {
 public:
  // declare a constructor
  SimpleString(char* cstr);

  // decare a destructor
  ~SimpleString();

  // member function (method)
  char& at(size_t index);

 private:
  char* m_data;
  size_t m_len;
};
```

I teach it this way because this is how objects are at their core. They are pretty much just structs that have a bit more.

They are laid out in memory the same way structs are

# Class vs Struct (Style)

- A class is the same thing as a struct. Only difference is if you don't specify an access modifier, class assumes private, structs assume public. THAT'S IT

- Our SimpleString (and Vec) would be better as a class stylistically.
- Typically, we only use and define struct's as we would in C: just a collection of data members (with only a few simple member functions if any)

- Common C++ style: prefix non-public data members with m_

```
class SimpleString {
 public:
  // declare a constructor
  SimpleString(char* cstr);

  // decare a destructor
  ~SimpleString();

  // member function (method)
  char& at(size_t index);

 private:
  char* m_data;
  size_t m_len;
};
```

# Synthesized Copying

- If you don't define the copy constructor, assignment operator, C++ will synthesize one for you
  - It will do a *shallow* copy of all of the data members (*i.e.* fields)
  - Sometimes the right thing; sometimes the wrong thing

Usually wrong whenever a class has dynamically allocated data

```
#include "SimpleString.hpp"

int main(int argc, char** argv) {
  SimpleString x;
  SimpleString y(x);
  y = x;              // invokes synthesized assignment operator
  return EXIT_SUCCESS;
}
```

# Synthesized Constructing/Destructing

- If you don’t define any constructor, C++ will synthesize one for you
  - Default initializes all member variables

- If you don’t define any destructor, C++ will synthesize one for you
  - Does nothing except call the destructor of any member variables

Usually wrong whenever a class has dynamically allocated data

# Explicitly ask for “default”

- If the synthesized constructor/destructor/copy constructor/assignment operator/etc. is something you want, then be explicit and declare them as “default”

```
class SimplePoint {
 public:
  SimplePoint() = default;
  ~SimplePoint() = default;
  SimplePoint(const SimplePoint other) = default;
  SimplePoint& operator=(const SimplePoint rhs) = default;
 private:
  int m_x;
  int m_y;
};
```

- Don’t have to implement them in the .cpp file if they are default
- Can also disable an operation by setting it `= delete;`

# Rule of three*

- If you want to define any of the following for a class:
  - Destructor
  - Copy constructor
  - Assignment operator

- It means there is some resource that needs to be properly handled/cleaned up, and you should implement them all (or disable some of them)

- This is really a rule of 5, but we haven't talked about the last two yet.

# Object Data Members

- What if we had a class like the following.
  What would the destructor for this class need to look like?
  What about the copy constructor?

```
class Course {
 public:
  // ...
 private:
  SimpleString m_dept;
  int m_number;
};
```

# Object Data Members

- What if we had a class like the following.
  What would the destructor for this class need to look like?
  What about the copy constructor?

```
class Course {
 public:
  // ...
 private:
  SimpleString m_dept;
  int m_number;
};
```

As long as SimpleString has the destructor, copy constructor, etc. properly defined, then the default destructor (empty destructor) and default copy constructor are fine.

# Regular Types

- A type is "regular" if it is
  - Copy constructable
  - Default (0-arg) constructable
  - Comparable (operator==)

- More emphasis on the first two: If you have those then using your newly defined object type will be much more compatible with the standard library and other libraries.

# That’s all for now!

- Still out:
  - HW00
  - HW01
  - Pre-semester Survey

- Hopefully you are doing well ☺