CMPT-101 Midterm

- It's in class, 50 minutes long
- It's open book
  - You can bring in *any* written notes or books
    - no sharing allowed in the exam
  - *No* electronics allowed!
- Bring your student card
  - Open book doesn't mean "don't study"
    - In fact, it's probably best to study for this exam as if it were a closed book exam

Write **Your** Midterm!

- You *must* write the midterm for the section you are *officially registered* for
  - check your registration if you are unsure
  - even if you think you are sure, check again!

Midterm Notes

- The best preparation is to practice writing and tracing programs
- There will definitely be some questions asking you to
  - *write* C++ code
  - *read* C++ code
- There *could* be short answer questions, multiple choice, true or false, fill in the blank, etc.

Midterm Covers Chapters 1-6

- Chapter 1, Introduction
- Chapter 2, Fundamental Data Types
- Chapter 3, Objects
- Chapter 4, Decisions
- Chapter 5, Functions
- Chapter 6, Iteration

Chapter 7: Testing and Debugging

- You should read this chapter on your own --- it is a very practical chapter that will help you when you write your own programs
- It won't be covered on this midterm, but it will be on the final!

Focus on the Essentials

- Using `ints`, `double`, `strings`, and variables
- Understand concepts like the type and scope of a variable
- Know how to use the graphics library
- if-else-if statements and boolean expressions

Focus on the Essentials

- Functions
  - Pass by value vs. pass by reference
  - Returning values
- Loops and recursion:
  - `while` loops, do-`while` loops, `for`-loops
  - the basic idea of recursion
- Know about documentation, program style, basic testing, algorithms, etc.

Lectures and Assignments

- Questions could be based on lecture material, assignments, the textbook, or the slides covered in class
  - Material covered in lectures follows the book, but with varying emphases and sometimes different examples

Do the Textbook Problems

- Solve (and bring!) answers to the textbook questions
  - Study groups are an excellent idea
- **Beware**: it takes time to look up answers in your notes, so you might not have time to check your notes for more than a couple of questions

Prepare a Study Guide

- Go through the textbook and all your notes and prepare a study guide emphasizing important points
  - Write examples of important concepts that you can read and understand at a glance

You can Write in Pen or Pencil

- However, if you write in pencil, we will not re-mark an exam if you think you've found a marking error
- The back of every exam page is blank, so you'll have lots of space to work out questions

Read the Mailing List

- If we decide to release any extra study materials, we will announce this to the mailing list
  - please don't email asking about old exams or answers to the textbook questions
  - if we make any available, we'll let everyone know via the mailing list

CMPT-101

Week 7

C++ Programmers Do it with `class`

What is a Class?

- A class is like a factory that creates objects
- A class is a *type*; an object is a *value*
- C++ lets you create your own classes
  - This is means C++ lets you create new types
  - This is a way of extending the language!

Object-oriented Design

- Classes are useful when you do object-oriented design (OOD)
- To do object-oriented design, you must first figure out what the objects and classes in the problem are
  - Easier said than done!
  - Many programmers find this to be the most interesting and challenging part of programming

Example: Bank Software

- Suppose you are writing the software for a bank
- Using object-oriented design, you must first "discover" the relevant classes
- You can do this by brain-storming, and listing common banking terminology
  - This is the first step of design, and things will be made more precise later

Bank Things

Customers, chequing accounts, savings accounts, money, tellers, ATMs, money, cheques, credit cards, bank cards, debit cards, PIN numbers, bills, interest, deposits, monthly statements, bank books, withdrawals, mutual funds, term deposits, stocks, bank books, safety deposit boxes, loans, mortgages, RRSPs, money orders, etc...

Bank Software

- The software for a bank could easily have classes for all the things mentioned on the previous slide
  - And more --- banks are relatively big, complex organizations
- Using classes that relate directly to every-day objects makes the software easier to write, maintain, extend, and understand

Bank Software

- Once you have some idea of the classes involved, you need to write the *interfaces* for these classes
  - that is, specify exactly what objects of each class can do, and what kind of information they store
  - e.g. a Customer has a name, has accounts, can transfer money between accounts, etc.

Bank Software

- When you've defined the interface for all your classes, you must then figure out how to combine them
  - this involves writing algorithms and functions that make all the objects work together in the right way

Writing Your Own Classes

- Classes consist of two things:
  - Member *variables*
  - Member *functions*

these are *owned* by objects

- Classes have two parts:
  - *Public* parts, accessible to all
  - *Private* parts, accessible only to the object's member functions

Classes can also have *protected* parts, but we won't get into that in this course.

Rule of Thumb

All* variables should be *private

- Public variables can changed by any function, and that makes it easy to accidentally corrupt an object
- It's usually best to make all variables private, and to write member functions that let you read and write their values

Private Variables

- In the textbook's `Time` class, you can only read the hours, minutes, and seconds
  - there's no way to set them after a `Time` object as been constructed!
- This is good design, and it means you can never have a `Time` object with an invalid time

A Class Template

```
class some_name {
public:
// ...

private:
// ...

}; // class some_name
```

Public variables and functions go here

Private variables and functions go here

Beware: forgetting this semi-colon often results in subtle, hard to find errors!

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);
private:
  string name_;
  Time start_;
}; // class TVshow
```

this is a *constructor*. A constructor is just a special member function that is called automatically when an object is created

Constructors *always* have

- the same name as the class
- no return type (they're not even `void`)

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);
private:
  string name_;
  Time start_;
}; // class TVshow
```

these are private member variables; only member functions in Tvshow can read/write `name_` and `start_`

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);
private:
  string name_;
  Time start_;
}; // class TVshow
```

the underscore at the end of the names is a useful convention --- it means that these are private member variables; it's not necessary, but it can make your code easier to read

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);
private:
  string name_;
  Time start_;
}; // class TVshow
```

Remember: a constructor is just a special kind of function that initializes an object, e.g. usually just giving default values to private variables

this is a function header for `TVshow`'s constructor --- no body has been defined yet; the constructor body must assign `name_` and `start_` their initial values

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);
private:
  string name_;
  Time start_;
}; // class TVshow

TVshow::TVshow(string name, Time start)
{ name_ = name;
  start_ = start;
}
```

this is how we can define the body `TVshow`'s constructor; notice that it's *outside* the class body

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);
private:
  string name_;
  Time start_;
}; // class TVshow

TVshow::TVshow(string name, Time start)
{ name_ = name;
  start_ = start;
}
```

the header for the constructor must be the same in both places, or you'll get a compiler error

A TV Show Class

the name of the class

```
// ... TVshow class is up here ...

TVshow::TVshow(string name, Time start)
{ name_ = name;
  start_ = start;
}
```

the name of a member function, which in this case is a constructor; a constructor *always* has the same name as the class

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);
private:
  string name_;
  Time start_;
}; // class Tvshow

TVshow::TVshow(string name, Time start)
{ name_ = name;
  start_ = start;
}
```

here's an example of how you create a `TVshow` object (e.g. in `main`, or some other function)

```
TVshow show("Friends",Time(20,0,0));
```

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);
private:
  string name_;
  Time start_;
}; // class Tvshow

TVshow::TVshow(string name, Time start)
{ name_ = name;
  start_ = start;
}
```

this automatically calls the constructor, which sets the name and the start time to the supplied values

```
TVshow show("Friends",Time(20,0,0));
```

A TV Show Class

```
TVshow show(”Friends”,Time(20,0,0));
```

This line creates a `TVshow` object and labels it `show`

`show`

`Name_: "Friends"`
`start_: Time(20,0,0)`

representation of a TVshow object in the computer's memory

A TV Show Class

```
TVshow show("Friends",Time(20,0,0));
```

That's all that `TVshow` can do right now --- it's not very useful! Now we want to add member functions that will let us set and read a `TVshow`'s attributes.

`show`

`Name_: "Friends"`
`start_: Time(20,0,0)`

Kinds of Member Functions

- Three main kinds of member functions have special names
  - **constructors** create and initialize objects
  - **accessors** read values of private variables
  - **mutators** change values of private variables

A Class Template

```
class some_name {
public:
// ... constructors come first ...

// ... accessors come second ...

// ... mutators come third ...
private:
// ... make all variables private...
}; // class some_name
```

Don't forget the semi-colon!

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);  // constructor
  // accessors
  string get_name() const;
  Time get_start() const;

  // mutators
  void set_name(string name);
  void set_start(Time start);
private:
  string name_;
  Time start_;
}; // class TVshow
```

by putting `const` here, we are telling the compiler that this function will not change the value of any member variables

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);  // constructor
  // accessors
  string get_name() const;
  Time get_start() const;

  // mutators
  void set_name(string name);
  void set_start(Time start);
private:
  string name_;
  Time start_;
}; // class TVshow
```

`const` comes after the function header, but before the semi-colon

A TV Show Class

```
class TVshow {
public:
  TVshow(string name, Time start);  // constructor
  // accessors
  string get_name() const;
  Time get_start() const;

  // mutators
  void set_name(string name);
  void set_start(Time start);
private:
  string name_;
  Time start_;
}; // class TVshow
```

Accessors should always be `const` like this --- it's not an accessor if you don't use `const`!

A TV Show Class

```
// ... TVshow class defined up here ...

string TVshow::get_name() const
{ return name_;
}
Time TVshow::get_start() const
{ return start_;
}
void TVshow::set_name(string name)
{ name_ = name;
}
void TVshow::set_start(Time start)
{ start_ = start;
}
```

the bodies of member functions are defined outside the class

notice that `TVshow::` appears before each function name; this is how C++ knows that these functions belong to class `TVshow`

A TV Show Class

```
// ... TVshow class defined up here ...

string TVshow::get_name() const
{ return name_;
}
Time TVshow::get_start() const
{ return start_;
}
void TVshow::set_name(string name)
{ name_ = name;
}
void TVshow::set_start(Time start)
{ start_ = start;
}
```

any member function of `TVshow` is allowed to access `name_` and `start_`

also, any member function is allowed to call any other member function in the same class, without using the dot notation

A TV Show Class

```
// ... TVshow class defined up here ...

int main()
{
  TVshow show("Friends", Time(20,0,0));
  cout << show.get_name() << “ is the name of the show.”
  cout << "\nIt’s starting time is:\n"
       << show.get_time().get_hours() << "hours\n"
       << show.get_time().get_minutes() << "minutes\n"
       << show.get_time().get_seconds() << "seconds\n";

  show.set_name("Enemies");
  cout << show.get_name() << " is the show’s new name";
}
```