CS31 MT1 Review Session

Sign in: https://goo.gl/QCeYby

Overview

- Libraries
- Types and Variables
- Basic Input/Output
- C++ Strings
- Control Flow (If/Else, Switches, Loops)
- Scoping

Libraries

- `#include` allows us to use a library
- `#include <iostream>` allows us to use things like:
  - `cin`
  - `cout`
  - `endl`
- *Note: `iostream` stands for input/output stream*

Namespaces

- `using namespace std;`
- A namespace is a collection of classes and functions
- If we don't call `using namespace` *ns_name*, we will have to specify the namespace of the function we want to call.
- e.g. `std::cout`, `std::string`, `std::isdigit`

Namespaces (cont.)

```
#include <iostream>

int main() {
  int age;
  std::cin >> age;
  std::cout << age;
  std::cout << std::endl;
}
```

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

int main() {
  int age;
  cin >> age;
  cout << age;
  cout << endl;
}
```

Basic data types

- `int, double, char`
  - Declare variables to store values in memory
  - `int x; // Creates a variable x of type int`
  - `char y; // Creates a variable y of type char`
- Can initialize with value at declaration:
  - `int a = 5;`
  - `double z = 53.24324;`

Modifying variables

The type of the variable must be specified only once, at the time of declaration

```
int x = 5;
x = x + 5;
x -= 6; // equivalent to x = x - 6;

double z = 53.234;
z *= 5; // equivalent to z = z * 5;
```

Modifying variables (cont.)

- Integer division truncates after the decimal point
- The % (modulus) operator returns the remainder of integer division

```
int x = 5;
int integerQuotient = x / 3; // integerQuotient equals 1
int remainder = x % 3;       // remainder equals 2
x %= 4;                      // same as x = x % 4, x now equals 1
```

Modifying variables (cont.)

- Double division
  - If at least one of the operands is a double, floating point division occurs.
  - If both values are integers, integer division occurs instead.

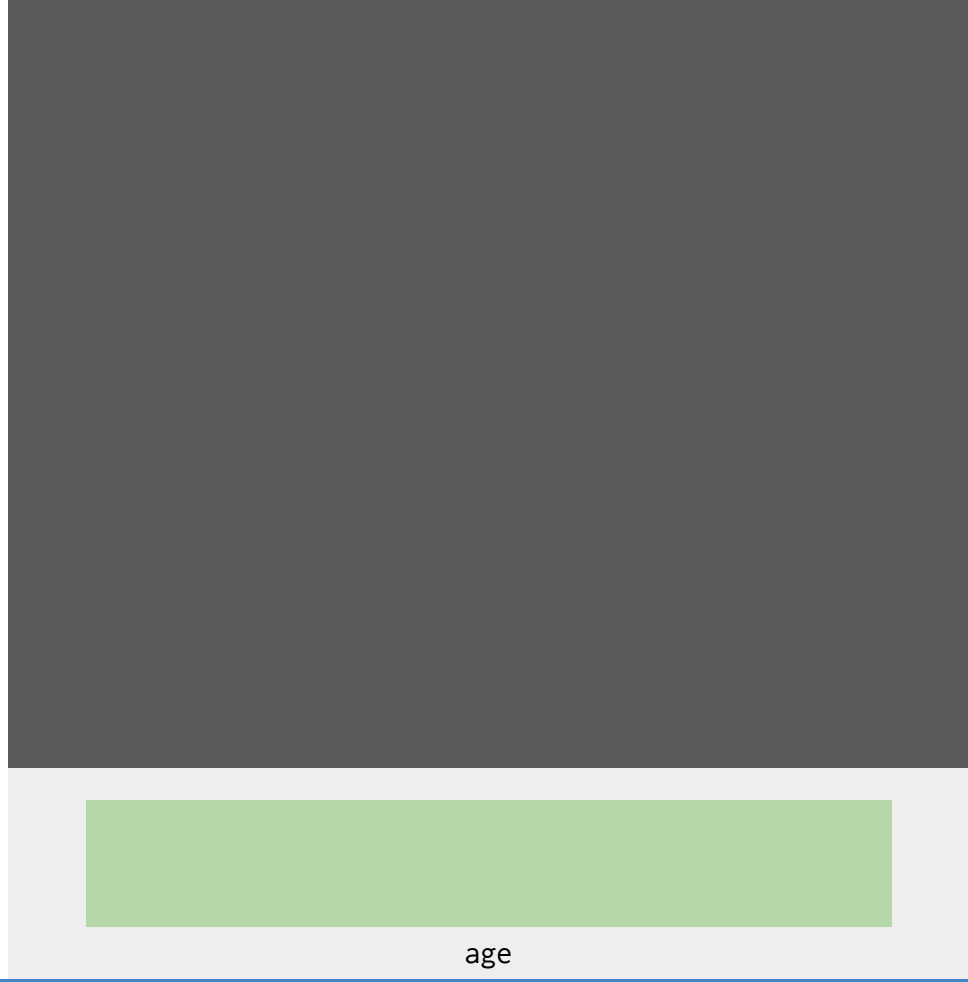
```
int x = 5;
double unexpectedQuotient = x / 2;    // equals 2.0
double expectedQuotient = x / 2.0;    // equals 2.5
```

Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```

Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```



Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```

```
> How old are you?
```

age

Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```

```
> How old are you? 20
```

20

age

Input/output

```
#include <iostream>
using namespace std;
int main() {
  int age;
  cout << "How old are you? " << endl;
  cin >> age;
  cout << "You are " << age <<
    " years old" << endl;
}
```

```
> How old are you? 20
> You are 20 years old
```

Strings

- Used to store blocks of text
- Strings can be initialized from literals
  - `string x = “hello”;`
- Individual characters can be accessed with the `[ ]` operator.
  - `char c = x[0]; // c == ‘h’`

String operations

```
string x = "hello there";
```

- The `size()` method returns the number of characters in a string.
  - `int length = x.size(); // length equals 11`
- The `substr(startIndex, length)` method returns a substring *including* `startIndex` of length `length`.
  - `string sub = x.substr(3, 2); // sub equals "lo"`
- *Note: `substr` is not in the scope of the midterm.*

String operations

```
// The + operator is overloaded:
// It appends to the end of strings.

int main() {
  string x = "hello there";
  x += ", my name is Mark";
  cout << x << endl;
}
```

hello there

x

String operations

```
// The + operator is overloaded:
// It appends to the end of strings.

int main() {
  string x = "hello there";
  x += ", my name is Mark";
  cout << x << endl;
}
```

hello there, my name is Mark

x

String operations

```
// The + operator is overloaded:
// It appends to the end of strings.

int main() {
  string x = "hello there";
  x += ", my name is Mark";
  cout << x << endl;
}
```

```
> hello there, my name is Mark
```

hello there, my name is Mark

x

String input

```
// getline(...) consumes characters from
// the input until it encounters a '\n'.

int main() {
  string x;
  getline(cin, x);
  cout << x << endl;
}
```

String input

```
// getline(...) consumes characters from
// the input until it encounters a '\n'.

int main() {
  string x;
  getline(cin, x);
  cout << x << endl;
}
```

```
> Why hello there!
```

String input

```
// getline(...) consumes characters from
// the input until it encounters a '\n'.

int main() {
  string x;
  getline(cin, x);
  cout << x << endl;
}
```

```
> Why hello there!
> Why hello there!
```

Ignoring characters

- Undesirable characters are often left in the input buffer after using cin.
- `cin.ignore(int numChars, char delim)` can be used to “flush” out these undesired characters. It flushes up to the nearest `delim` or `numChar` characters, whichever comes first.
- `cin.ignore(...)` becomes necessary if after reading a number, the next thing you want to read is a string using `getline(...)`.

Ignoring characters

- Undesirable characters are often left in the input buffer after using cin.
- `cin.ignore(int numChars, char delim)` can be used to “flush” out these undesired characters. It flushes up to the nearest `delim` or `numChar` characters, whichever comes first.
- `cin.ignore(...)` becomes necessary if after reading a number, the next thing you want to read is a string using `getline(...)`.

- **Common question**: does `getline` consume `'\n'`?

Ignoring characters

- Undesirable characters are often left in the input buffer after using cin.
- `cin.ignore(int numChars, char delim)` can be used to “flush” out these undesired characters. It flushes up to the nearest `delim` or `numChar` characters, whichever comes first.
- `cin.ignore(...)` becomes necessary if after reading a number, the next thing you want to read is a string using `getline(...)`.

- **Common question**: does `getline` consume `'\n'`?

  If a newline is found, it is extracted and discarded (i.e. it is not stored and the next input operation will begin after it).

cin.ignore example

```
int main() {
  cout << "How many Big Macs would you " <<
      "like? ";
  int bigMacs;
  cin >> bigMacs;
  cin.ignore(10000, '\n'); // Important!

  cout << "What else would you like " <<
    "with your order?";
  string sides;
  getline(cin, sides);
}
```

```
> How many Big Macs would you like?
```

Input

cin.ignore example

```
int main() {
  cout << "How many Big Macs would you " <<
      "like? ";
  int bigMacs;
  cin >> bigMacs;
  cin.ignore(10000, '\n'); // Important!

  cout << "What else would you like " <<
    "with your order?";
  string sides;
  getline(cin, sides);
}
```

```
> How many Big Macs would you like?
```

bigMacs

Input

cin.ignore example

```
int main() {
  cout << "How many Big Macs would you " <<
      "like? ";
  int bigMacs;
  cin >> bigMacs;
  cin.ignore(10000, '\n'); // Important!

  cout << "What else would you like " <<
    "with your order?";
  string sides;
  getline(cin, sides);
}
```

```
> How many Big Macs would you like? 1000
```

bigMacs	Input
1000	\n

cin.ignore example

```
int main() {
  cout << "How many Big Macs would you " <<
      "like? ";
  int bigMacs;
  cin >> bigMacs;
  cin.ignore(10000, '\n'); // Important!

  cout << "What else would you like " <<
    "with your order?";
  string sides;
  getline(cin, sides);
}
```

```
> How many Big Macs would you like? 1000
```

bigMacs

Input

cin.ignore example

```
int main() {
  cout << "How many Big Macs would you " <<
      "like? ";
  int bigMacs;
  cin >> bigMacs;
  cin.ignore(10000, '\n'); // Important!

  cout << "What else would you like " <<
    "with your order?";
  string sides;
  getline(cin, sides);
}
```

```
> How many Big Macs would you like? 1000
> What else would you like with your
  order?
```

bigMacs	Input
1000	

cin.ignore example

```
int main() {
  cout << "How many Big Macs would you " <<
      "like? ";
  int bigMacs;
  cin >> bigMacs;
  cin.ignore(10000, '\n'); // Important!

  cout << "What else would you like " <<
    "with your order?";
  string sides;
  getline(cin, sides);
}
```

```
> How many Big Macs would you like? 1000
> What else would you like with your
  order?
```

bigMacs	sides	Input
1000		

cin.ignore example

```
int main() {
  cout << "How many Big Macs would you " <<
      "like? ";
  int bigMacs;
  cin >> bigMacs;
  cin.ignore(10000, '\n'); // Important!

  cout << "What else would you like " <<
    "with your order?";
  string sides;
  getline(cin, sides);
}
```

```
> How many Big Macs would you like? 1000
> What else would you like with your
  order? Fries
```

1000	Fries	
bigMacs	sides	Input

cin.ignore(...) example

- What will be stored in the string “a” in this example?
- Assume that input is newline terminated.

```
int x; string a;
cout << "Enter an integer" << endl;
cin >> x; // Assume the user enters “7”
cout << "Enter a string" << endl;
getline(cin, a); // Assume user enters “500”
```

cctype

- #include<cctype> gives you...
  - `isalpha('M')  // true, since 'M' is a letter`
  - `isupper('M')  // true, since 'M' is an uppercase letter`
  - `islower('r')  // true, since 'r' is a lowercase letter`
  - `isdigit('5')  // true, since '5' is a digit character`
  - `islower('M')  // false, since 'M' is not a lowercase letter`
  - `isalpha(' ')  // false, since ' ' is not a letter`
  - `isalpha('5')  // false, since '5' is not a letter`

Control flow

- `if` statements only run code if the condition is true
- *Note: any non-zero expression is considered true*

```
int age;
cin >> age;
if (age < 13) {
  cout << "You are not yet a teenager!" << endl;
}
```

Comparison pitfalls

- **Equals-equals (==) vs. Equals (=)**
- These operators are very different!

```
(x == y) // Returns true if x and y are equal

(x = y)  // Assigns the value of y to x and returns the value
         // ASSIGNED to x.
```

Conditional confusion?

Does this output anything?

```
int age = 17;
if (age) {
  cout << "You are not 0 years old!" << endl;
}
```

Conditional confusion?

What does this output?

```
int age = 0;
if (age) {
  cout << "You are not 0 years old!" << endl;
} else {
  cout << “You are 0 years old!” << endl;
}
```

If statements

- Without curly braces, only next statement is attached to the control statement.
- So, if you want multiple statements to be executed, use curly braces.
- *Note: this also applies to `else` and `else-if` statements*

```
if (cond1) {
  statement1;
  statement2;
}
```

If statements (cont.)

```
int main() {
  int x = 3;
  if (x == 5)
    cout << "x is 5" << endl;
  cout << "In if" << endl; // Incorrectly
                           // called!
}
```

```
int main() {
  int x = 5;
  if (x == 5) {
    cout << "x is 5" << endl;
    cout << "In if" << endl;
  }
}
```

Else statements

Performed when all `if` and `else if` conditions fail

```
int number;
cin >> number;
if (number % 2 == 0)
    cout << "You gave an even number" << endl;
else
    cout << "You gave an odd number" << endl;
```

Else-if

Allows us to check for more than the `if` condition and its complement

```
if (cond1)
    statement1;
else if (cond2)
    statement2;
else if (cond3)
    statement3;
else
    statement4;
```

Switches

- Arguably a more compact alternative to long `if/else if/else` sequences
- The value tested must be an integral type or convertible to one
  - e.g. `int`, `char`, `short`, `long`, etc.
  - `string` is not a permitted type
- A `break` statement must be used to leave the switch. Otherwise execution will fall through to the next case.

Switches (cont.)

```
string value; int number;
cin >> number;
switch (number) {
  case 0:  // Fall-through to Case 2.
  case 2:
    value = "Good";
    break; // Remember to break!
  case 3:
    value = "Bad";
    break;
  default:
    value = "Ugly";
}
```

Switches (cont.)

```
string value; int number;
cin >> number;
switch (number) {
  case 0:  // Fall-through to Case 2.
  case 2:
    value = "Good";
    break; // Remember to break!
  case 3:
    value = "Bad";
    break;
  default:
    value = "Ugly";
}
```

Common question: is the break statement required for the default case?

Switches (cont.)

```
string value; int number;
cin >> number;
switch (number) {
  case 0:  // Fall-through to Case 2.
  case 2:
    value = "Good";
    break; // Remember to break!
  case 3:
    value = "Bad";
    break;
  default:
    value = "Ugly";
}
```

Common question: is the break statement required for the default case?

Not necessarily, but we recommend that you do. This allows the default case to appear in a different order, not necessarily at the end of the switch statement.

While loops

```
int main() {
  int x = 0;
  while (x < 2) {
    cout << x << endl;
    x++;
  }
  cout << "Done!" << endl;
}
```

While loops

```
int main() {
  int x = 0;
  while (x < 2) {
    cout << x << endl;
    x++;
  }
  cout << "Done!" << endl;
}
```

```
> 0
```

0

x

While loops

```
int main() {
  int x = 0;
  while (x < 2) {
    cout << x << endl;
    x++;
  }
  cout << "Done!" << endl;
}
```

```
> 0
```

1

x

While loops

```
int main() {
  int x = 0;
  while (x < 2) {
    cout << x << endl;
    x++;
  }
  cout << "Done!" << endl;
}
```

```
> 0
```

1

x

While loops

```
int main() {
  int x = 0;
  while (x < 2) {
    cout << x << endl;
    x++;
  }
  cout << "Done!" << endl;
}
```

```
> 0
> 1
```

1

x

While loops

```
int main() {
  int x = 0;
  while (x < 2) {
    cout << x << endl;
    x++;
  }
  cout << "Done!" << endl;
}
```

```
> 0
> 1
```

2

x

While loops

```
int main() {
  int x = 0;
  while (x < 2) {
    cout << x << endl;
    x++;
  }
  cout << "Done!" << endl;
}
```

```
> 0
> 1
```

2

x

While loops

```
int main() {
  int x = 0;
  while (x < 2) {
    cout << x << endl;
    x++;
  }
  cout << "Done!" << endl;
}
```

```
> 0
> 1
> Done
```

2

x

Do-while loops

Same as while loops, except the first iteration always runs.

```
statement1;
do {
  statement2;
} while (cond1); // Don't forget the semicolon!

statement3;
```

For loops

- Declaration is run once before anything else
- Condition is evaluated before the code block is executed
- Action is run after the code block is executed

```
for (declaration; condition; action) {
  statement1;
  statement2;
}
```

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  // Note that i is now out of scope.
  cout << "Done!" << endl;
}
```

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  // Note that i is now out of scope.
  cout << "Done!" << endl;
}
```

0

i

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  // Note that i is now out of scope.
  cout << "Done!" << endl;
}
```

```
> i is now equal to: 0
```

0

i

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  // Note that i is now out of scope.
  cout << "Done!" << endl;
}
```

```
> i is now equal to: 0
```

1

i

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  // Note that i is now out of scope.
  cout << "Done!" << endl;
}
```

```
> i is now equal to: 0
```

1

i

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  // Note that i is now out of scope.
  cout << "Done!" << endl;
}
```

```
> i is now equal to: 0
> i is now equal to: 1
```

1

i

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  // Note that i is now out of scope.
  cout << "Done!" << endl;
}
```

```
> i is now equal to: 0
> i is now equal to: 1
```

2

i

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  // Note that i is now out of scope.
  cout << "Done!" << endl;
}
```

```
> i is now equal to: 0
> i is now equal to: 1
```

2

i

For loops

```
int main() {
  for (int i = 0; i < 2; i++) {
    cout << "i is now equal to: " << i <<
        endl;
  }

  cout << "Done!" << endl;
}
```

```
> i is now equal to: 0
> i is now equal to: 1
> Done!
```

Nested loops

```
for (int i = 1; i <= 10; i++) {
  for (int j = 1; j <= 10; j++)  {
    cout << (i * j) << "\t";
  }
  cout << endl;
}
```

Quick Question

- What happens when you `break` inside nested loops?
  - Only the loop that contains the `break` statement is broken out of.

Quick Question

- What happens when you `break` inside nested loops?
  - Only the loop that contains the `break` statement is broken out of.

- Solutions

Quick Question

- What happens when you `break` inside nested loops?
  - Only the loop that contains the `break` statement is broken out of.

- Solutions
  a. Can use `goto` statement to break out of nested loops completely, but this is unadvised (can cause problems if used incorrectly/carelessly).

Quick Question

- What happens when you `break` inside nested loops?
  - Only the loop that contains the `break` statement is broken out of.

- Solutions
  a. Can use `goto` statement to break out of nested loops completely, but this is unadvised (can cause problems if used incorrectly/carelessly).
  b. Can use a boolean variable in each loop statement and change the boolean from true to false when you want to break out of all nested loops.

Scoping

Variables only exist within the curly brackets or the implied curly brackets that they were written in.

```
if (cond1) {
  statement1;
}
```

Scoping (cont.)

```
if (cond1) {
  int x = 5;
  cout << x << endl; // No error
}

cout << x << endl; // Error!! x doesn't exist
                   // outside the if statement
```

Scoping (cont.)

```
int x = 1;
if (cond1) {
  x = 5;
  cout << x << endl; // No error
}

cout << x << endl; // No error here either!
```

Scoping (cont.)

```
string s1 = “bonjour”;
for (int i = 0; i < s1.size(); i++) {
  char lastChar = s1[i];
}

// Both i and lastChar don't exist here!
cout << i << " " << lastChar << endl; // Error!
```

Scoping (cont.)

```
string s1 = “bonjour”;
int i; char lastChar;
for (i = 0; i < s1.size(); i++) {
  lastChar = s1[i];
}

// Now both i and lastChar exist here
cout << i << " " << lastChar << endl;
```

Need more help?

- UPE offers daily tutoring for all lower division classes!
  - Location: **ACM/UPE Clubhouse (2763 BH)**
  - Schedule: **https://upe.seas.ucla.edu/tutoring/**

- UPE's Google-sponsored CS31 Trivia Night
  - Wednesday 11/2, 6:15-7:30pm, Boelter 4760
  - Free exclusive Google swag!!

- Practice problems & solutions: https://goo.gl/DFmyrH
  - Warning: can contain stuff from the next midterm too

Good luck!

Slides: https://goo.gl/W2CScG
Facebook Event: https://goo.gl/zieTp2
Sign in: https://goo.gl/QCeYby

Any last quick questions? Come up and ask us! We'll try our best.
Any last long questions? Post on the Facebook event page!