CS 1428 Review

CS 2308 :: Spring 2016
Molly O’Neil
Structure of a C++ Program

• **Hello world**

```cpp
// This program prints a greeting to the screen
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

• **Generalized**

```cpp
// This is a comment
#include <includefile1>
using namespace std;

void functionPrototypes(argument-types, ...);

int main() {
    statements ... 
    return 0;
}

void functionDefinition(arguments, ...) {
    statements ...
}
```
Variables & Identifiers

• **Variable**: a storage location in the computer’s memory

• **Identifier**: the name of a program element (e.g., a variable). Must conform to some rules:
  - First character must be in a–z or A–Z or _
  - Other characters must in a–z or A–Z or _ or 0–9
  - Cannot be a reserved C++ keyword (see Gaddis Table 2-4)
  - Case sensitive: myinteger is not the same as myInteger
  - Check class style guidelines for more info

• All variables must be **declared** (and can optionally be **initialized** in the same statement):

```cpp
int number;         // Variable Declaration
char letter;       // stores a letter grade

int number = 42;   // life, the universe, and everything
char letter = 'Q'; // Variable Initialization
```
Fundamental Data Types

- **Integers**

<table>
<thead>
<tr>
<th>short int</th>
<th>int</th>
<th>long long int</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short int</td>
<td>unsigned int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>usually 2 bytes</td>
<td>usually 4 bytes</td>
<td>usually 8 bytes</td>
</tr>
<tr>
<td>s: -32,768 to +32,767</td>
<td>s: -2b to +~2 billion</td>
<td>s: -9q to +~9 quintillion</td>
</tr>
<tr>
<td>u: 0 to +65,535</td>
<td>u: 0 to +~4 billion</td>
<td>u: 0 to +~18 quintillion</td>
</tr>
</tbody>
</table>

- **Floating-Point (Real) Numbers**

<table>
<thead>
<tr>
<th>float</th>
<th>double</th>
<th>long double</th>
</tr>
</thead>
<tbody>
<tr>
<td>usually 4 bytes</td>
<td>usually 8 bytes</td>
<td>usually 8-10 bytes. if 10:</td>
</tr>
<tr>
<td>+/-3.4e-38 to +/-3.4e38</td>
<td>+/-1.7e-308 to +/-1.7e308</td>
<td>+/-3.4e-4932 to +/-3.4e4932</td>
</tr>
</tbody>
</table>

- **Characters**

<table>
<thead>
<tr>
<th>char</th>
</tr>
</thead>
</table>

- **Boolean values (can only hold true or false)**

<table>
<thead>
<tr>
<th>bool</th>
</tr>
</thead>
</table>
Strings

• Not a built-in/fundamental data type, but included in standard C++

• Must `#include <string>` for most operations

```cpp
string firstName, lastName;
```

• Operations include:
  • `=` for assignment
  • `.size()` member function for character-count
  • `==, <, ...` relational operators (alphabetical order, within case)
  • `[n]` for access to single char
Literals & Named Constants

- **Literal**: a specific value written directly into a program’s code

  - 1
  - 75
  - -12
  - 11.42
  - -3.8
  - 6.25e-5
  - 'A'
  - '2'
  - '
  - true
  - false
  - "This is my string I want printed to the screen"

- **Named Constants**: a variable whose value cannot be changed

  ```
  const datatype identifier = constant;
  ```

  ```
  const double TAX_RATE = 0.0825;
  ```

- Use these often in place of literals! Good code has very few “magic constants”
Assignment

- Modify the value stored in a variable (copy in a new value)

```
variable = expression;
```

```
count = 10;
```

- The left-hand side must be an lvalue (storage location)
- The right-hand-side is an expression that yields the correct type
- An expression has a type and evaluates to a value
  - Literal
  - Named constant
  - Variable
  - Arithmetic expression

```
sum = 5;
sum = INITIAL_SUM;
sum = initCount;
sum = 1 + 2;
sum = initCount + newCount;
```
Operators

- Negation (unary)
  - - negation

- Arithmetic (binary)
  - + addition
  - - subtraction
  - * multiplication
  - / division
  - % modulo

- Relational (binary) — result is always a bool
  - == equals
  - != not equal to
  - > greater than
  - < less than
  - >= greater than or equal to
  - <= less than or equal to
Operators

- Logical - result is always a `bool`
  - `!` not [unary]
  - `&&` and [binary]
  - `||` or [binary]

- Compound Assignment (binary)
  
<table>
<thead>
<tr>
<th>operator</th>
<th>e.g.</th>
<th>equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>x += e;</code></td>
<td><code>x = x + e;</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>x -= e;</code></td>
<td><code>x = x - e;</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>x *= e;</code></td>
<td><code>x = x * e;</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>x /= e;</code></td>
<td><code>x = x / e;</code></td>
</tr>
</tbody>
</table>

- Increment/Decrement (unary)

<table>
<thead>
<tr>
<th>operator</th>
<th>e.g.</th>
<th>equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code></td>
<td><code>x++;</code></td>
<td><code>x = x + 1;</code></td>
</tr>
<tr>
<td><code>--</code></td>
<td><code>x--;</code></td>
<td><code>x = x - 1;</code></td>
</tr>
</tbody>
</table>

- Conditional (ternary)

```
expression ? expression : expression;
```

```
y = x < 0 ? 10 : 20;
```
Operator Precedence

• Answers “which happens first?”

- (unary)
  * / %
  + - (binary)
  < > <= >=
  == !=
  &&
  ||

• Parentheses can override operator precedence
  • When in doubt, use parentheses (within reason)

```c
!(y == 10) || y == 20 && x > 3 * -z + 2
!(y == 10) || ( y == 20 && x > (3 * -z + 2) )
!(y == 10) || ( (y == 20) && ( x > ( (3 * (-z)) + 2 ) ) )
```

parentheses used to override precedence:
```c
( !(y == 10) || y == 20 ) && x > 3 * (-z + 2)
```
Type Conversion

- **Type coercion**: If you use an operator on two different data types, C++ will *implicitly* (try to) convert operands to same type
  - `char, short` always promoted to `int`
  - When operator works on 2 types, lower rank promoted to higher
  - If expression yields value of one type that is assigned to lvalue of other type, it is converted to the type of the lvalue
    - **Caution: this may result in loss of information!**

```
int x, y = 2;
float z = 1.2;
x = y * z;       // x = ??
```

- **Type casting**: You can also manually *(explicitly)* promote/demote a value

```
int sum, count;
...
float avg = static_cast<float>(sum) / count;
```
User-Defined Data Types (Structs)

- A **struct** (structure) stores a user-defined collection of physically-grouped variables of possibly-different types
- *Define* the new struct data type, then *declare* variables of your new type
- Each element in the struct is a member
  - Accessed using the . (dot) operator

```c
struct Student {
    int idNumber;
    string name;
    float gpa;
};
```

```c
Student student1, student2;
Student student3 = {123456, "Alan Turing", 3.57};
student1.name = "Ada Lovelace";
student2.idNumber = 987654;
```
Struct Operations

• **Valid** operations over entire structs:
  • Assignment: `student1 = student2;`
  • Function call: `myFunc(student1, x);`

• **INVALID** operations over structs:
  • Comparison: `student1 == student2`
  • Output: `cout << student1;`
  • Input: `cin >> student2;`
  • All of these must be done member by member
I/O

• \#include <iostream>

• Output (\texttt{cout} and \texttt{<<})

```c++
cout << expression;
cout << expr1 << expr2;

cout << "Hello world!\n";
cout << "Count is: " << count << endl;
```

• Input (\texttt{cin} and \texttt{>>})

```c++
cin >> variable;  \textbf{RHS must be an lvalue}
cin >> var1 >> var2;

cin >> x;
cout << "Enter the height and width: ";
cin >> height >> width;
```
Output Formatting

• `#include <iomanip>`

• `setw`: sets field width for immediately-following value

```cpp
int value = 23;
cout << "( " << setw(5) << value << " )";
```

• `setprecision`: sets the # of significant digits of FP #

```cpp
double value = 28.92786;
cout << setprecision(3) << value;
```

• `fixed`: sets fixed-point notation. With fixed, setprecision sets # of digits to right of decimal point

```cpp
double value = 28.92786;
cout << setprecision(2) << fixed;
cout << value;
```

• `showpoint`: shows trailing zeros for existing setprecision, always shows decimal point for FP

```cpp
double value = 32.0;
cout << setprecision(5) << showpoint << value;
```

• `left / right`: left or right justify text inside field width
File I/O

• #include <fstream>

• Output (ofstream)

```cpp
ofstream fOut;
fOut.open("filename.txt");
fOut << "Hello!\n";
fOut << "Count is: " << count << endl;
fOut.close();
```

• Input (ifstream)

```cpp
ifstream fIn;
fIn.open("filename.txt");
if(!fIn) {
    cout << "Error opening file" << endl;
    return -1;
}

int num;
fIn >> num;  // RHS must be an lvalue
cout << "You entered: " << num << endl;
fIn.close();
```

Always check for file errors!
File I/O: Reading to EOF

- `fIn >> x` returns true when a value is successfully read, false otherwise

```cpp
ifstream fIn;
fIn.open("input.txt");

int x;
while(fIn >> x) {
    cout << "Next number is: " << x << endl;
}

fIn.close();
```

- Final time `fIn >> x` is attempted, nothing is read into `x`, expression returns false
I/O: Dealing with Spaces

- >> operator uses whitespace as delimiter, leading to problems when a token has more than one word (e.g., a full name)

- Solution: `getline(fileStream, str)`
  - Reads an entire line (to a newline character) from the specified file stream (either cin or an ifstream object)
  - Stores the data read from the file stream to string 'str'

```cpp
string input;
ifstream nameFile;
nameFile.open("names.txt");
ggetline(nameFile, input);
```

- When used in combination with >>, must ignore leading whitespace characters left by >> operator, e.g.:

```cpp
cin.ignore(100, '\n'); // ignores the next 100 characters or // until newline encountered
```
Decision/Control Structures

- **Compound statements ("block")**

```java
{  
    statement1;
    statement2;
    ...
}
```

- **Procedural programming**: list of instructions (often separated into subroutines) specifying what computer should do, step by step, from first instruction to next

  - **Sequence structure**: statements executed in sequence, without any choices of direction (branches)

  - **Decision structure**: allows some statements to be executed only under certain conditions

  ![Decision flowchart](image)
If/Else Statements

- If/Else
  - If expression is true (non-zero), statement_block1 is executed
  - If expression is false, statement_block2 is executed

- The “else” part is optional

- Nested if/else
Switch Statements

- The value of a variable/expression determines where of several options the program will branch

```cpp
switch(expression) {
    case constant1: statements
    case constant2: statements
    ...
    default: statements < default case is optional
}
```

- Execution starts at case labeled with expression’s value
- If no match, execution starts at default
- Use `break;` to exit a switch case (otherwise execution continues through following cases)

```cpp
switch(ch) {
    case 'a':
    case 'A': cout << "You chose A";
                break;
    case 'b':
    case 'B': cout << "You chose B";
                break;
    default: cout << "Invalid choice";
}
```
Loops

• While loop

```cpp
int x = 0;
while(x <= 0) {
    cout << "Enter a positive integer: ";
    cin >> x;
}
```

• if expression is true, statement_block is executed; repeat

• Do-while loop

```cpp
int x;
do {
    cout << "Enter a positive integer: ";
    cin >> x;
} while(x <= 0);
```

• For loop

```cpp
for(int i = 0; i < 10; i++) {
    // statements here
}
```

Equivalent to:

```cpp
init_expr;
while(test_expr) {
    statements
    update_expr;
}
```
Terminating a Loop Iteration

- Sometimes you want to exit a loop iteration before the end of the loop body. 2 statements to do so:
  - **break**: Terminates both the loop iteration and the loop itself, i.e., the condition will not be re-checked and execution will proceed from the closing curly brace that ends the loop scope.
  - **continue**: Skips the rest of the loop body and continues with the next loop iteration (after re-checking the loop entry condition).

```c++
while(true) {
    char x;
    cout << "enter e to exit or c to continue: ";
    cin >> x;

    if(x == 'e')
        break;
    if(x == 'c')
        continue;

    cout << "you entered something other than e or c!" << endl;
}
cout << "exited." << endl;
```
Nested Loops

• Lots of problems require nesting one loop inside another

```cpp
for(int hour = 0; hour < 24; hour++) {
    for(int minute = 0; minute < 60; minute++) {
        for(int second = 0; second < 60; second++) {
            cout << hour << "h ";
            cout << minute << "m ";
            cout << second << "s" << endl;
        }
    }
}
```

$ ./a.out

0h 0m 0s
0h 0m 1s
...
0h 0m 59s
0h 1m 0s
...
0h 59m 59s
1h 0m 0s
...
23h 59m 59s
Functions

- **Function**: collection of statements that perform specific, related task
  - Divide problem into small, manageable pieces
  - Allow for easy *code reuse*
- Function Definition:

  ```
  datatype identifier (parameter1, parameter2, ...) {
    statements ...
  }
  ```

  - `datatype`: the data type returned by the function
  - `identifier`: the name by which the function can be called
  - `parameters`: allow passing arguments to the function when it is called; from within the function, act as a local variable
  - `statements`: The function’s body; executed (with parameter values from call) when function is called


Functions

- **Function call**: `identifier(expression1, ...)`
  - Causes control flow to branch to body of function named `identifier`
  - `parameter1` in function definition initialized to value of `expression1`, etc.

- **Return statement**: `return expression;
  - Inside a function, causes function to stop & return control to caller
  - Value of return `expression` (if present) becomes value of functional call expression
Functions

- **Void function**: a function that returns no value
  - Initialization routines, display routines, etc.

```cpp
void printSum(int a, int b) {
    int result = a + b;
    cout << "The result is " << result << endl;
}

type main() {
    printSum(2, 7);
}
```

- **Function prototype**: placed before a function call to tell compiler details about fn’s return type & parameters

  ```cpp
datatype identifier(datatype1, datatype2, ...);
```

  - Alternative is to place entire function definition before first call to function
  - You can give parameter identifiers as well, but optional
Pass by Value vs. Pass by Reference

- **Pass by value**: when an argument is passed to a function, its *value is copied* behind-the-scenes into the parameter
  - Parameter and argument are separate storage locations
  - Changes to parameter in function body do NOT change value of the argument in the call
  - This is what happens by default in C++ (except for arrays, sort of... stay tuned)

- **Pass by reference**: when an argument is passed to a function, the function has *direct access to the original argument*
  - Single storage location (reference parameter acts as *alias* to argument)
  - Changes to the parameter in function body DO change value of the argument in the call
  - Implemented in C++ using reference parameters (add & after parameter type in function prototype and definition)

```cpp
void zeroMyValue(int &myValue) {
    myValue = 0;
}
```
#include <iostream>
using namespace std;

void zeroMyValue(int);

int main() {
    int myValue = 42;
    cout << "Inside main: myValue = " << myValue << endl;
    zeroMyValue(myValue);
    cout << "Back inside main: ";
    cout << "myValue = " << myValue << endl;
    return 0;
}

void zeroMyValue(int myValue) {
    myValue = 0;
    cout << "Inside zeroMyValue: ";
    cout << "myValue = " << myValue << endl;
}
Pass by Reference Example

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

void zeroMyValue(int &);

int main() {
    int myValue = 42;
    cout << "Inside main: myValue = " << myValue << endl;
    zeroMyValue(myValue);
    cout << "Back inside main: ";
    cout << "myValue = " << myValue << endl;
    return 0;
}

void zeroMyValue(int &myValue) {
    myValue = 0;
    cout << "Inside zeroMyValue: ";
    cout << "myValue = " << myValue << endl;
}
```

What is the program output now?
Variable Scope

• A variable's **scope** defines in which parts of the program the variable can be accessed/"seen"
  - Variables exist only inside their own scope, destroyed at the end of it
• **Global variable** (defined outside of all functions): accessible everywhere after its definition
• **Local variable** (defined inside a function): accessible inside the block `{ }` in which it is defined, after its definition
• **Parameter**: accessible anywhere inside of its function body
• Variables at the exact same scope cannot have the same name
• Variable defined in inner block will block a variable with same name in outer block

```cpp
int x = 10;
{
    int x = 20;
    cout << x << endl;
}
cout << x << endl;
```

outputs: 20 10
Arrays

• An array is:
  • A series of values of the same data type
  • Stored in contiguous memory locations
  • That can be individually referenced by index

• Declaration:
  
  \[
  \text{datatype identifier[SIZE];}
  \]

• \textit{datatype} is the type of the array values
• \textit{identifier} is the name of the array
• \textit{SIZE} is the number of elements (literal or constant)

• Initialization:

  \[
  \text{float scores[3] = \{86.5, 92.1, 77.5\};}
  \]

  // initializes first 3 indices, zeroes the rest

  \[
  \text{float scores[10] = \{86.5, 92.1, 77.5\};}
  \]

  // determines array size from init list (3)

  \[
  \text{float scores[] = \{86.5, 92.1, 77.5\};}
  \]
Array Elements & Indexing

• To access the value of a particular array element:

```c
scores[2] = 77.5;
```

• If array `scores` is an array of floats, `scores[2]` is a float
  • Can be used like any other float variable:
    • as an lvalue (i.e., you can assign a new value to it)
    • in arithmetic expressions
    • passed as an argument to a function

• Subscripts/indexes always begin at 0, end at `arraySize-1`

• Array index must have type `int`, but can be the result of any expression that yields an `int`

• Generally must operate on individual array elements, not entire array, e.g.:
  • Cannot assign one array to another
  • Cannot input directly into an array
  • Cannot use relational operators to compare one array to another
Passing Arrays to Functions

- In the function prototype:
  ```
  void showArray(int [], int);
  ```

- In the function definition:
  ```
  void showArray(int arr[], int size) { ... }
  ```

- In the function call, pass the array name (no brackets!)
  ```
  int numbers[5] = {1, 2, 3, 4, 5};
  showArray(numbers, 5);
  ```

- In C++, arrays are implicitly passed by reference
  - If you pass an array to a function, function can modify the original array in the calling scope
  - `const` keyword in front of type in parameter declaration to disallow
Array Function Example

• Find sum of all elements in an array

```c
int sumArray(const int [], int);

int main() {
    const int NUM_ELEMENTS = 10;
    int numbers[NUM_ELEMENTS];
    ... // assign values to numbers[0] through numbers[9]
    int sum;
    sum = sumArray(numbers, NUM_ELEMENTS);
}

int sumArray(const int arr[], int size) {
    int total = 0;
    for(int i = 0; i < size; i++) {
        total += arr[i];
    }
    return total;
}
```

• Other common array functions: find largest element, find smallest element, count occurrences of a value, etc.
Parallel Arrays vs. Arrays of Structs

• To store related data of different types, you could keep multiple arrays with shared subscripts

```c
int hoursWorked[5];
double payRate[5];
...
double employee0GrossPay = hoursWorked[0] * payRate[0];
```

• Or you could use an Array of Structures

```c
struct Employee {
    int hoursWorked;
    double payRate;
};
...
Employee employees[5];
...
double employee0GrossPay = employees[0].hoursWorked * employees[0].payRate;
```
2-Dimensional Arrays

- You can think of a 2-D array as several 1-D arrays of the same size and type strung together into consecutive memory

- Definition:
  ```
  double scores[NROWS][NCOLS];
  ```

- Element access:
  ```
  scores[row_num][col_num];
  ```

- Nested loops and 2-D array iteration:
  ```
  for(int r = 0; r < NROWS; r++) {
      for(int c = 0; c < NCOLS; c++) {
          cout << "scores[" << r << "][" << c << "] = ";
          cout << scores[r][c] << endl;
      }
  }
  ```
Default Arguments

• **Default argument**: value passed to a parameter automatically if no argument is provided in function call
  - Assigned in earliest occurrence of function name (usually this is the fn prototype)

```cpp
void printArea(int = 10, int = 5);

void printArea(int a, int b) {
    cout << "The area is " << a * b << endl;
}

void printArea(int a = 10, int b = 5) {
    cout << "The area is " << a * b << endl;
}
```

**If there's no function prototype**, then default args go in function definition:

```cpp
void printArea(int a = 10, int b = 5) {
    cout << "The area is " << a * b << endl;
}
```

• **Function can be called as follows:**

```cpp
printArea(2, 6);  // outputs: "The area is 12"
printArea();      // outputs: "The area is 50"
printArea(3);     // outputs: "The area is 15"
```
Default Argument Rules

• If you leave an argument out of a function call, all arguments that come after it must also be left out:

```c
printArea( , 1);  // DOES NOT COMPIL
```

• If not all parameters to a function have default argument values, the parameters with defaults must come last:

```c
void printArea(int, int = 5);  // OK
void printArea(int = 10, int);  // NOT OK
```
Function Overloading

- Two (or more) functions may have the same name, as long as their parameter lists are different
  - Different types and/or number of parameters
  - Compiler determines which function to call by matching arguments # and type(s) in call to parameter lists

```c++
int square(int number) {
    return number * number;
}

double square(double number) {
    return number * number;
}

int main() {
    cout << square(10) << "\n";
    cout << square(1.2) << "\n";
}

double calcWeeklyPay(int hours, double payRate) {
    return hours * payRate;
}

double calcWeeklyPay(double salary) {
    return salary / 52;
}
```