Introduction to C

CS 2060 Week 1

Prof. Jonathan Ventura

Why C Programming?

- Developed in 1970s by Dennis Ritchie at Bell Labs
- Used to re-write UNIX
- Still one of the most popular programming languages
- A low-level "high-level" language – fast and small code, close-to-metal
- Lacks features of later languages like type safety, objects

Difference from other languages

Java	C
(Almost) everything is a class	No concept of classes
Compiles to byte code	Compiles to machine code
Runs in virtual machine	Runs directly on microprocessor
Standardized across platforms	Varies across platforms

Table: Some differences between Java and C

Relevance of C today

- Google "Why C Programming" to find articles like:
  - "Why C Programming Still Runs the World"
  - "Why C and C++ are Awful Programming Languages"
- Many strong opinions about C either way
- Still an essential skill for computer scientists
- Gateway to other languages like C++
- Not the best for prototyping
- Great for writing highly-optimized, processor-specific code

Course Links

- Course webpage: http://jventura.net/cs2060
- Blackboard: http://bb.uccs.edu
  - Grades will be managed in Blackboard
- Moodle: https://gvg.uccs.edu/moodle
  - Programming assignments administered in Moodle
    - **https** required
    - Google Chrome, Firefox supported; Safari doesn't work
    - Need to accept my self-signed certificate

Course Objectives

- Write and compile programs in the C language
- Understand structured programming (functions and loops)
- Use the C standard library
- Understand memory allocation and management (stack and heap)
- Manage memory using pointers
- Create and use arrays
- Perform file I/O
- Create and manipulate strings
- Use bitwise operators
- Implement simple data structures such as a linked list

Course Structure

- Reading assignments: about one chapter per week
  - Textbook: C How to Program 8/e, Deitel and Deitel
- Programming assignments: due Mondays (11:59 PM)
- Quizzes: Thursday at start of class
- Midterm
- Final

Grading

Grade Distribution:

Category	Points Possible	Percentage of Grade
Quizzes	100	10%
Homeworks	300	30%
Midterm Exam	200	20%
Final Exam	400	40%
Total	1000	100%

Class Policies

- Computers and phones are not to be used in class except for taking notes.
- Quizzes and exams are closed book, closed notes.

Class Policies

- Students are expected to work independently. **Offering** and **accepting** solutions from others is an act of **plagiarism**, which is a serious offense and **all involved parties will be penalized according to the Academic Honesty Policy**. Discussion amongst students is encouraged, but when in doubt, direct your questions to the professor, tutor, or lab assistant.
- Copying and pasting code from the Internet is plagiarism and is not allowed in this course – even if you modify what you copy. Work your own solutions to the assigned problems and come to me and/or the CS tutors for help.

Class Policies

- Attendance is expected.
- Students are responsible for all missed work, regardless of the reason for absence. It is also the absentee's responsibility to get all missing notes or materials.
- Late homeworks will not be accepted.
- Makeup quizzes will not be given.
  - This is because the homeworks and quizzes will be explained immediately after they are due.
- Makeup exams will only be arranged with consent of the instructor given **prior** to the exam day.

Resources

- CS Tutors in the Math Tutoring Center
- My office: ENGR 194
- My office hours: W 2-4 (or just email me jventura@uccs.edu)

Hello World in C

```
// helloworld.c
#include <stdio.h>

int main( void )
{
        printf("Hello, World!\n");

        return 0;
}
```

Syntax observations:

- (Most) lines must end with a semicolon.
- Code blocks are encapsulated in curly brackets { }.
- Whitespace does not matter and is basically ignored.
- // marks a single-line comment.

Hello World in C

```c
// helloworld.c
#include <stdio.h>

int main( void )
{
        printf("Hello, World!\n");

        return 0;
}
```

Structure observations:

- Every C program contains a `main()` function.
- `main()` (mostly) describes the entire execution of the program.
- `void` is C keyword meaning nothing – `main` has no arguments here.

Hello World in C

```c
// helloworld.c
#include <stdio.h>

int main( void )
{
        printf("Hello, World!\n");

        return 0;
}
```

About `#include`:

- `#include` is a preprocessor command.
- Copies `stdio.h` directly into the source code.
- `stdio.h` is part of the C standard library;
  it provides the `printf()` function.

Hello World in C

What does the code do??

Let's find out!!

`https://gvg.uccs.edu/moodle`

C Compilation Process

Compiling a C executable consists of several steps:

1. Preprocessor
   - Resolves macro commands like `#include`
2. Compiler
   - Translates C source files into object files (machine code)
   - Links to external functions are left as placeholders
3. Linker
   - Connects links between object code and external libraries

C Compilers

- GCC (Gnu Compiler Collection) is a widely used open-source compiler
- Available on Linux, Mac, Windows (through MinGW or Cygwin)
  - Mac now uses LLVM compiler; calling `gcc` will actually just invoke LLVM.
- From command line:
  - `gcc -o myprog myprog.c`
  - `./myprog`
- Windows: DevCPP IDE is available on virtual machines
- **However:** learning command line hacking is highly recommended!

Errors in the C Coding Process

Possible error instances:

1. Compile-time errors, e.g.:
   - Syntax error
   - Incorrect function arguments
2. Linker
   - Expected external functions are not there – need to include library
3. Run-time errors, e.g.:
   - Divide by zero
   - Segmentation fault (trying to read from wrong part of memory)
   - Out of memory

Adding two integers

```c
// addition.c
#include <stdio.h>

int main( void )
{
  // read two integers
  int a;
  int b;
  scanf( "%d", &a );
  scanf( "%d", &b );

  // add them together
  int sum;
  sum = a + b;

  // print the result
  printf( "%d\n", sum );
}
```

Variable definitions:

- Variables must be defined before they are used.
- Variables must be given a name and a type, e.g.
  `int` for integers
  `float` for reals
- Names are case-sensitive.
- Name cannot start with a number.

Adding two integers

```c
// addition.c
#include <stdio.h>

int main( void )
{
  // read two integers
  int a;
  int b;
  scanf( "%d", &a );
  scanf( "%d", &b );

  // add them together
  int sum;
  sum = a + b;

  // print the result
  printf( "%d\n", sum );
}
```

`scanf` parses text input.

- `scanf` means "**f**ormatted **scan**"
- The first argument is the format specifier: `%d` indicates one integer will be read.
- The following arguments say where values will be stored.
- The & operator returns a **pointer** to the variable.
- The pointer allows `scanf` to modify the variable's value.

Adding two integers

```c
// addition.c
#include <stdio.h>

int main( void )
{
  // read two integers
  int a, b;
  scanf( "%d %d", &a, &b );

  // add them together
  int sum;
  sum = a + b;

  // print the result
  printf( "%d\n", sum );
}
```

Multiple values in one call to `scanf`:

- We can easily read two integers in one `scanf` call.
- `scanf` will keep reading whitespace until it finds the second value.
- Note that we can also define two variables on the same line.

Adding two integers

```c
// addition.c
#include <stdio.h>

int main( void )
{
  // read two integers
  int a, b;
  scanf( "%d %d", &a, &b );

  // add them together
  int sum;
  sum = a + b;

  // print the result
  printf( "%d\n", sum );
}
```

Arithmetic operators:

- We can easily add variables using the + operator.
- The result is stored in a third variable.
- `sum = a + b` will not modify `a` or `b`.

Adding two integers

```c
// addition.c
#include <stdio.h>

int main( void )
{
  // read two integers
  int a, b;
  scanf( "%d %d", &a, &b );

  // add them together
  int sum;
  sum = a + b;

  // print the result
  printf( "%d\n", sum );
}
```

Let's try running it.

Adding two integers

```c
// addition.c
#include <stdio.h>

int main( void )
{
  // read two integers
  int a, b;
  int nread = scanf( "%d %d", &a, &b );
  if ( nread != 2 )
  {
    printf("Error\n");
    return 1;
  }

  // ....
}
```

Checking for input errors:

- scanf returns the number of values read.
- If bad input is given, it will stop parsing values –
- for example, if text is input when a number is expected.

Adding two integers

```c
// addition.c
#include <stdio.h>

int main( void )
{
  // read two integers
  int a, b;
  int nread = scanf( "%d %d", &a, &b );
  if ( nread != 2 )
  {
    printf("Error\n");
    return 1;
  }

  // ....
}
```

Control logic:

- `if` indicates a conditional block.
- The condition must be wrapped in parentheses.
- `!=` is the inequality operator.

Modulo operator

```
#include <stdio.h>

int main()
{
  int a, b;
  scanf( "%d %d", &a, &b );

  int a_div_b = a / b;
  int a_mod_b = a % b;
  printf( "%d = %d * %d + %d\n", ??? );
}
```

- % is the *modulo* or remainder operator.
- `a % b` gives the remainder of `a / b`.
- What should go in the ??? to make a true statement?

Modulo operator

```
#include <stdio.h>

int main()
{
  int a, b;
  scanf( "%d %d", &a, &b );

  int a_div_b = a / b;
  int a_mod_b = a % b;
  printf( "%d = %d * %d + %d\n",
          a, a_div_b, b, a_mod_b );
}
```

- When operating on integers, a/b drops the remainder.
- `a == ( a / b ) * b + a % b`

Order of Arithmetic Operations

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first, inner to outer.
*	Multiplication	Evaluated second, left to right.
/	Division	
%	Modulo	
+	Addition	Evaluated third, left to right.
-	Subtraction	
=	Assignment	Evaluated last.

Table: Precedence of arithmetic operators in C.

Arithmetic puzzles

What does this evaluate to?

```
1 + 2 + 3 + 4 + 5 / 3
```

Arithmetic puzzles

What does this evaluate to?

```
1 + 2 + 3 + 4 + 5 / 3
```

```
= 11
```

Arithmetic puzzles

What does this evaluate to?

```
(1 + 2 + 3 + 4 + 5) / 3
```

Arithmetic puzzles

What does this evaluate to?

```
(1 + 2 + 3 + 4 + 5) / 3
```

```
= 5
```

Arithmetic puzzles

What does this evaluate to?

```
10*10/5
```

Arithmetic puzzles

What does this evaluate to?

```
10*10/5
```

= 20

Arithmetic puzzles

What does this evaluate to?

```
10*10/3
```

Arithmetic puzzles

What does this evaluate to?

```
10*10/3
```

= 33

Evaluated left to right: 10*10 is evaluated first.

Arithmetic puzzles

What does this evaluate to?

```
10*(10/3)
```

Arithmetic puzzles

What does this evaluate to?

```
10*(10/3)
```

= 30

Parentheses (10/3) evaluated first.

Arithmetic puzzles

What does this evaluate to?

```
10*10/3/2
```

Arithmetic puzzles

What does this evaluate to?

```
10*10/3/2
```

= 16

Evaluated left to right.

Arithmetic puzzles

What does this evaluate to?

```
10*10+3/2
```

Arithmetic puzzles

What does this evaluate to?

```
10*10+3/2
```

= 101

Multiplication and division before addition or subtraction.

Arithmetic puzzles

What does this evaluate to?

```
10*(10+3)/2
```

Arithmetic puzzles

What does this evaluate to?

```
10*(10+3)/2
```

= 65

Parentheses first.

Comparing Integers

```c
#include <stdio.h>

int main()
{
  int a, b;
  scanf( "%d %d", &a, &b );

  if ( a > b ) printf( "a > b\n" );
  else if ( a < b ) printf( "a < b\n" );
  else printf( "a == b\n" );
}
```

Control logic:

- else and else if used for false case.

Comparing integers

```
#include <stdio.h>

int main()
{
  int a, b;
  scanf( "%d %d", &a, &b );

  if ( a == b ) printf( "equal\n" );
  else printf( "not equal\n" );
}
```

Comparison operators:

- == for equality
- != for inequality
- >= for greater-than-or-equal-to
- <= for less-than-or-equal-to

Comparing integers

```
#include <stdio.h>

int main()
{
  int a, b;
  scanf( "%d %d", &a, &b );

  if ( a = b ) printf( "equal\n" );
  else printf( "not equal\n" );
}
```

What is the mistake here?

Comparing integers

```c
#include <stdio.h>

int main()
{
  int a, b;
  scanf( "%d %d", &a, &b );

  if ( a = b ) printf( "equal\n" );
  else printf( "not equal\n" );
}
```

- A common mistake is using = instead of ==.
- `a = b` will return the value of `a` after assignment.
- Any non-zero value evaluates to true, and zero evaluates to false.
- So, this code will print “equal” if `b != 0`.

Comparing integers

```
#include <stdio.h>

int main()
{
  int a, b;
  scanf( "%d %d", &a, &b );

  if ( ( a = b ) ) printf( "b != 0\n" );
  else printf( "b == 0\n" );
}
```

- Modern compilers will give a warning if they suspect a mistake with = instead of ==.
- Wrap the assignment in parentheses to suppress the warning.

Order of Operations

Operator(s)	Associativity
( )	left to right
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
=	right to left

Table: Precedence and associativity of operators discussed so far.

Associativity

```c
#include <stdio.h>

int main()
{
  int a = 5;
  int b = 2;
  int c = 3;

  a = b = c;
}
```

- Operator associativity determines how operators of the same precedence are grouped.
- Operators discussed so far are left-to-right, except for assignment.
- Can make multiple assignments in one statement.

- `b = c` is evaluated first; evaluates to new value of `b`.
- `a = ...` is evaluated second, setting `a` to value of `b = c`.

Homework 1

Part 1: Statistics

- Description:
  - Create a program which reads three integers from the console and outputs the following:

```
Minimum value:  <value>
Maximum value:  <value>
Median value:  <value>
```

  - Don't forget a final carriage return "`\n`" at the end.
  - If bad input is given, the program should output "error" and exit.
- Rules:
  - Do not sort the numbers – no swapping.
  - No `for` loops.
  - Use only `if`, `else`, and the comparison operators.

Optional: use the **minimum** worst-case number of comparisons.

Counting comparisons

```c
#include <stdio.h>

// output yes if input in range [33...66]
int main()
{
  int a;
  scanf( "%d", &a );

  if ( a < 33 ) printf("no\n");
  else if ( a >= 33 && a <= 66 ) printf("yes\n");
  else if ( a > 66 ) printf("no\b");
}
```

- This program tests whether the input value is in the range [33, ..., 66].
- How many comparisons does it use in the worst case?

Counting comparisons

```c
#include <stdio.h>

// output yes if input in range [33...66]
int main()
{
  int a;
  scanf( "%d", &a );

  if ( a < 33 ) printf("no\n");
  else if ( a >= 33 && a <= 66 ) printf("yes\n");
  else if ( a > 66 ) printf("no\b");
}
```

- This program tests whether the input value is in the range [33, ..., 66].
- How many comparisons does it use in the worst case?
- Four, if a is greater than 66.

Counting comparisons

```c
#include <stdio.h>

// output yes if input in range [33...66]
int main()
{
  int a;
  scanf( "%d", &a );

  if ( a < 33 ) printf("no\n");
  else if ( a <= 66 ) printf("yes\n");
  else printf("no\b");
}
```

- We can reduce the number of comparisons but have the same result.
- How many comparisons does this program use in the worst case?

Counting comparisons

```c
#include <stdio.h>

// output yes if input in range [33...66]
int main()
{
  int a;
  scanf( "%d", &a );

  if ( a < 33 ) printf("no\n");
  else if ( a <= 66 ) printf("yes\n");
  else printf("no\b");
}
```

- We can reduce the number of comparisons but have the same result.
- How many comparisons does this program use in the worst case?
- Two, if a is greater than 66.

Homework 1

Part 2: Counting polynomial roots

- Description:
  - Accept three integer coefficients a, b, c of a quadratic polynomial:
    $ax^2 + bx + c = 0$
  - Output the number of real-valued roots of the polynomial.
  - If bad input is given, the program should output "error" and exit.

Homework 1

Part 2: Counting polynomial roots

- The quadratic formula gives the roots to the polynomial: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- $d = b^2 - 4ac$ is called the discriminant.
  - If $d > 0$, the polynomial has two real roots.
  - If $d == 0$, the polynomial has one (repeated) real root.
  - if $d < 0$, the polynomial has zero real roots.

C Coding Style

- Writing obscure, arcane code can be a joy :)
  - International Obfuscated C Contest
  - Self-modifying code
- However, adopting a coding style will help make your programs more readable, more usable and less error-prone.
- Writing understandable code is important, even if your only collaborator is yourself.
  - You don't want to come back to your code one year later and ask, "wait, what does this do???!"

Google Style Guide

- Google maintains coding style guides for many languages:
  - Google C++ Style Guide
- "As every C++ programmer knows, the language has many powerful features, but this **power** brings with it **complexity**, which in turn can make code **more bug-prone** and **harder to read and maintain**.
  The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing C++ code. These rules exist to keep the code base manageable while still allowing coders to use C++ language features productively."

Google Style Guide: Variable Names

- “Names should be descriptive; eschew abbreviation.
- Give as descriptive a name as possible, within reason.
- Do not worry about saving horizontal space
  - as it is far more important to make your code immediately understandable by a new reader.
- Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project,
- and do not abbreviate by deleting letters within a word.”

Google Style Guide: Variable Definition

```c
// Bad -- initialization separate from declaration.
int i;
i = f();

// Good -- declaration has initialization.
int j = g();
```

- Try to initialize variables immediately.
- Avoids mistakenly using un-initialized variable.

Google Style Guide: Variable Names

```c
// Good examples:
int price_count_reader;     // No abbreviation.
int num_errors;             // "num" is a widespread convention.
int num_dns_connections;    // Most people know what "DNS" stands for.

// Bad examples:
int n;                      // Meaningless.
int nerr;                   // Ambiguous abbreviation.
int n_comp_conns;           // Ambiguous abbreviation.
int wgc_connections;        // Only your group knows what this stands for.
int pc_reader;              // Lots of things can be abbreviated "pc".
int cstmr_id;               // Deletes internal letters.
```

Premature Optimization

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; **premature optimization is the root of all evil** (or at least most of it) in programming."
– Donald Knuth

Premature Optimization

- Before you try to make your code fast, make sure it works correctly!
- Optimization typically makes your code difficult to understand.
- Performance optimization should be reserved for only the truly critical parts of your code.
- Shorter code **does not mean** faster code.

Developing a personal coding style

```
int numerator = 0;
int denominator = 0;

int num_read = scanf( "%d %d",
                      &numerator, &denominator );
if ( num_read != 2 )
{
  printf("error: couldn’t read two integers\n");
  return 1;
}
```

Syntax choices:

- Spaces around parentheses for readability.
- Brackets on their own lines.

Developing a personal coding style

```
int numerator = 0;
int denominator = 0;

int num_read = scanf( "%d %d",
                      &numerator, &denominator );
if ( num_read != 2 )
{
  printf("error: couldn't read two integers\n");
  return 1;
}
```

Testing results of functions:

- Avoid putting functions inside a conditional.
- Store the result in a variable for readability.

Developing a personal coding style

```
if ( denominator == 0 )
{
  printf( "error: divide by zero\n" );
}
else
{
  int fraction = numerator/denominator;
  printf( "%d / %d = %d\n",
          numerator, denominator, fraction );
}
```

Block structures:

- Put one-liners in a code block.
    - Easier to parse structure of code.
    - Faster when you need to add extra lines later.
- Follow the pattern of brackets on their own lines, even for if/else.

Developing a personal coding style

```
if ( denominator == 0 )
{
  printf( "error: divide by zero\n" );
}
else
{
  int fraction = numerator/denominator;
  printf( "%d / %d = %d\n",
          numerator, denominator, fraction );
}
```

Comments:

- This code is simple and self-explanatory enough;

  comments not really needed here.

Next week

Next Week:

- Homework 1 due Monday, 11:55 PM
  - Last saved version of your code will be tested.
- Lecture topic: C Program Control (`if`, `else`, `while`)
- Quiz on Thursday: Covering all material up to then