

# Procedure Programming

## Introduction and Hello World

Instructor: Jeeho Ryoo

# A little background about me

Call me *Jeeho*!

Education:

ECE at Cornell

MS/PhD at Univ of Texas at Austin

Work Experience:

Senior Software Engineer at ***Oracle***

Senior System Engineer at ***ARM***

# Contact Info

Contact me:

Via email: [jee\\_ho\\_ryoo@bcit.ca](mailto:jee_ho_ryoo@bcit.ca)

Office hours:

By appointment only (max flexibility)  
*Over zoom. Link posted on LH*

# Evaluation Criteria

Criteria	%	Comments
Labs	20	9
Quizzes	10	3
Assignments	20	3
Midterm Exam	20	
Final Exam	30	Cumulative

# Evaluation Criteria

## Quizzes:

- During lecture using Github

- Live coding

- Leetcode style

# Evaluation Criteria

## Labs:

No late submissions

All done in C

Released Tuesday and due on Friday

# Evaluation Criteria

## Assignments:

You would be working on an assignment on most weeks.

Assignments are usually due in ~1.5-2 weeks (10-14 days):

Team of max 4

# Evaluation Criteria

Midterm and Final exams:

- All paper based

- Whiteboard coding

- Multiple choice

- Some output questions



- **For major assessments:** A doctors note or picture proof of covid-19 positive test is required for accommodation.

- This would apply for all major assessment that cannot be accommodated virtually such as final exams and midterms.
- Picture proof should include the student ID and proof of date. An example of proof of date would be an article from the CBC with the date visible in the background.

- **For everything else:** We are going to be lenient this term again. No doctors note or proof of covid test will be required as per institute policy. Students are to discuss individual accommodations, if any, with their instructor and inform them if they can't make it to class.

- Instructors can choose to accommodate sick students virtually if they wish to do so but this is not a requirement.

# Goals

- ▶ C syntax
- ▶ Standard libraries
- ▶ Programming for robustness and speed
- ▶ Understanding compiler

# Structure of a C Program

## Overall Program

<some pre-processor directives>

<global declarations>

<global variables>

<functions>

# Structure of a C Program

## Overall Program

<some pre-processor directives>

<global declarations>

<global variables>

<functions>

## Functions

<function header>

<local declarations>

<statements>

# hello.c: Hello World

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf("Hello World\n");
```

```
    return 0;
```

```
}
```

# Compiling and Running

- ▶ `$ gcc hello.c -o hello`

- ▶ `$ ./hello`

Hello World

# What Happens?

- ▶ `$ gcc hello.c -o hello`
  - ▶ Compile “hello.c” to machine code named “hello”
  - ▶ “-o” specifies the output file name. (Notice it’s case-sensitive.)
- ▶ `$ ./hello`
  - ▶ Execute program “hello”
  - ▶ “./” is necessary!

# What Happens?

- ▶ `$ gcc hello.c -o hello`
  - ▶ Compile “hello.c” to machine code named “hello”
  - ▶ “-o” specifies the output file name. (Notice it’s case-sensitive.)
- ▶ `$ ./hello`
  - ▶ Execute program “hello”
  - ▶ “./” is necessary!

## hello.c

```
#include <stdio.h> // "printf" is declared in this header file.

int main() // Main point of execution.
{
    printf("Hello World\n"); // Output "Hello World" to console.
    return 0; // Tell OS the program terminates normally.
}
```



## vars.c: Variables

```
#include <stdio.h>

int main()
{
    int a, b, c;

    a = 10;
    b = 20;
    c = a * b;
    printf("a = %d b = %d c = %d\n", a, b, c);
    return 0;
}
```

## vars.c: Variables

```
#include <stdio.h>

int main()
{
    int a, b, c;

    a = 10;
    b = 20;
    c = a * b;
    printf("a = %d b = %d c = %d\n", a, b, c);
    return 0;
}
```

a = 10 b = 20 c = 200

## cmdarg.c: Command Line Args

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int n, m;

    n = atoi(argv[1]);
    m = atoi(argv[2]);
    printf("Argument 1: %d\nArgument 2: %d\n", n, m);
    return 0;
}
```

## cmdarg.c: Command Line Args

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int n, m;

    n = atoi(argv[1]);
    m = atoi(argv[2]);
    printf("Argument 1: %d\nArgument 2: %d\n", n, m);
    return 0;
}
```

\$ ./cmdarg 10 20

Argument 1: 10

Argument 2: 20

## More on printf

- ▶ `printf(format_string, val1, val2);`

# More on printf

- ▶ `printf(format_string, val1, val2);`
  - ▶ `format_string` can include placeholders that specify how the arguments `val1`, `val2`, etc. should be formatted
  - ▶ `%c` : format as a character
  - ▶ `%d` : format as an integer
  - ▶ `%f` : format as a floating-point number
  - ▶ `%%` : print a `%` character

# More on printf

- ▶ `printf(format_string, val1, val2);`
  - ▶ `format_string` can include placeholders that specify how the arguments `val1`, `val2`, etc. should be formatted
  - ▶ `%c` : format as a character
  - ▶ `%d` : format as an integer
  - ▶ `%f` : format as a floating-point number
  - ▶ `%%` : print a `%` character

## Examples

```
float f = 0.95;  
printf("f = %f%%\n", f * 100);
```

# More on printf

- ▶ `printf(format_string, val1, val2);`
  - ▶ `format_string` can include placeholders that specify how the arguments `val1`, `val2`, etc. should be formatted
  - ▶ `%c` : format as a character
  - ▶ `%d` : format as an integer
  - ▶ `%f` : format as a floating-point number
  - ▶ `%%` : print a `%` character

## Examples

```
float f = 0.95;  
printf("f = %f%%\n", f * 100);
```

f = 95.000000%



# More on printf

- ▶ Placeholders can also specify widths and precisions
  - ▶ `%10d` : add spaces to take up at least 10 characters
  - ▶ `%010d` : add zeros to take up at least 10 characters
  - ▶ `%.2f` : print only 2 digits after decimal point
  - ▶ `%5.2f` : print 1 decimal digit, add spaces to take up 5 chars

# More on printf

- ▶ Placeholders can also specify widths and precisions
  - ▶ %10d : add spaces to take up at least 10 characters
  - ▶ %010d : add zeros to take up at least 10 characters
  - ▶ %.2f : print only 2 digits after decimal point
  - ▶ %5.2f : print 1 decimal digit, add spaces to take up 5 chars

## Examples

```
float f = 0.95;
printf("f = %.2f%%\n", f * 100);
// f = 95.00%
printf("f = %10.2f%%\n", f * 100);
// f =      95.00%
```

# Warning about printf

- ▶ printf is powerful, but potentially dangerous

What does this code output?

```
int i = 90;
float f = 3;
printf("f = %f i = %d\n", f);
printf("f = %f\n", f, i);
printf("i = %d f = %f\n", f, i);
```

# Statements

**<statement> := <expression>;**

```
x = 0;
```

```
++i;
```

```
printf("%d", x);
```

# Blocks

`<block> := {<statements>}`

```
{  
x = 0;  
++i;  
printf("%d", x);  
}
```

# Blocks

$\langle \text{block} \rangle := \{ \langle \text{statements} \rangle \}$

```
{  
x = 0;  
++i;  
printf("%d", x);  
}
```

- A block is syntactically equivalent to a single statement.

# Blocks

`<block> := {<statements>}`

```
{  
x = 0;  
++i;  
printf("%d", x);  
}
```

- ▶ A block is syntactically equivalent to a single statement.
  - ▶ if, else, while, for
  - ▶ Variables can be declared inside *any* block.
  - ▶ There is no semicolon after the right brace that ends a block.

## Example

```
int x = 0;
{
    int x = 5;
    printf("Inside: x = %d\n", x);
}
printf("Outside: x = %d\n", x);
```



## Example

```
int x = 0;
{
    int x = 5;
    printf("Inside: x = %d\n", x);
}
printf("Outside: x = %d\n", x);
```

Inside: x = 5

Outside: x = 0

# if Statement

**if (<condition>) <statement>**

```
// single statment
```

```
if (2 < 5)
    printf("2 is less than 5.\n");
```

```
// block
```

```
if (2 < 5)
{
    printf("I'll always print this line.\n");
    printf("because 2 is always less than 5!\n");
}
```

# if-else Statement

if (<condition>) <statement1> else <statement2>

```
if (x < 0)
{
    printf("%d is negative.\n", x);
}
else
{
    printf("%d is non-negative.\n", x);
}
```

# else-if Statement

```
if (a < 5)
    printf("a < 5\n");
else
{
    if (a < 8)
        printf("5 <= a < 8\n");
    else
        printf("a >= 8\n");
}
```

```
if (a < 5)
    printf("a < 5\n");
else if (a < 8)
    printf("5 <= a < 8\n");
else
    printf("a >= 8\n");
```

# if-else Statement Pitfalls

```
if (a > 70)
    if (a > 80)
        printf("grade = B\n");
else
    printf("grade < B\n");
    printf("Fail.\n");
printf("Done.\n");
```

```
if (a > 70)
{
    if (a > 80)
    {
        printf("grade = B\n");
    }
    else
    {
        printf("grade < B\n");
    }
}
printf("Fail.\n");
printf("Done.\n");
```

# Relational Operators

C has the following relational operators

<code>a == b</code>	true iff a equals b
<code>a != b</code>	true iff a does not equal b
<code>a &lt; b</code>	true iff a is less than b
<code>a &gt; b</code>	true iff a is greater than b
<code>a &lt;= b</code>	true iff a is less than or equal to b
<code>a &gt;= b</code>	true iff a is greater than or equal to b
<code>a &amp;&amp; b</code>	true iff a is true and b is true
<code>a    b</code>	true iff a is true or b is true
<code>!a</code>	true iff a is false

# Booleans in C

- ▶ C DOES NOT have a boolean type.
- ▶ Instead, conditional operators evaluate to integers (int)
  - ▶ 0 indicates false. Non-zero value is true.
  - ▶ if (<condition>) checks whether the condition is non-zero.

# Booleans in C

- ▶ C DOES NOT have a boolean type.
- ▶ Instead, conditional operators evaluate to integers (int)
  - ▶ 0 indicates false. Non-zero value is true.
  - ▶ if (<condition>) checks whether the condition is non-zero.
  - ▶ Programmer must be very careful to this point!



# Booleans in C

- ▶ C DOES NOT have a boolean type.
- ▶ Instead, conditional operators evaluate to integers (int)
  - ▶ 0 indicates false. Non-zero value is true.
  - ▶ if (<condition>) checks whether the condition is non-zero.
  - ▶ **Programmer must be very careful to this point!**

## Examples

```
if (3)
    printf("True.\n");

if (!3)
    // unreachable code

if (a = 5)
    // always true, potential bug (a == 5)

int a = (5 == 5); // a = 1
```

# Conditional expressions

`<condition> ? <expression1> : <expression2>`

```
grade = (score >= 70 ? 'S' : 'U');
```

```
printf("You have %d item%s.\n", n, n == 1 ? "" : "s");
```

# Conditional expressions

`<condition> ? <expression1> : <expression2>`

```
grade = (score >= 70 ? 'S' : 'U');
```

```
printf("You have %d item%s.\n", n, n == 1 ? "" :  
      "s");
```

Conditional expression often leads to succinct code.

# switch Statement

## A common form of if statement

```
if (x == a)
    statement1;
else if (x == b)
    statement2;
...
else
    statement0;
```

# switch Statement

## A common form of if statement

```
if (x == a)
    statement1;
else if (x == b)
    statement2;
...
else
    statement0;
```

## switch statement

```
switch (x)
{
    case a: statement1; break;
    case b: statement2; break;
    ...
    default: statement0;
}
```

# More on switch Statement

## Fall-through property

```
int month = 2;

switch (month)
{
    case 1:
        printf("Jan.\n");
        break;
    case 2:
        printf("Feb.\n");
    case 3:
        printf("Mar.\n");
    default:
        printf("Another month.\n");
}
```

# More on switch Statement

## Fall-through property

```
int month = 2;

switch (month)
{
    case 1:
        printf("Jan.\n");
        break;
    case 2:
        printf("Feb.\n");
    case 3:
        printf("Mar.\n");
    default:
        printf("Another month.\n");
}
```

Feb.  
Mar.  
Another month.

# More on switch Statement

## Fall-through property

```
int month = 2;
int days;

switch (month)
{
    case 2:
        days = 28;
        break;
    case 9:
    case 4:
    case 6:
    case 11:
        days = 30;
        break;
    default:
        days = 31;
}
```



# More on switch Statement

## Fall-through property

```
int month = 2;
int days;

switch (month)
{
    case 2:
        days = 28;
        break;
    case 9:
    case 4:
    case 6:
    case 11:
        days = 30;
        break;
    default:
        days = 31;
}
```

It's always recommended to have **default**, though it's optional.

# while Loop

- ▶ while (<condition>) <statement>

# while Loop

- ▶ while (<condition>) <statement>
  - ▶ If the condition is initially false, the statement is never executed.

# while Loop

- ▶ while (<condition>) <statement>
  - ▶ If the condition is initially false, the statement is never executed.
- ▶ do <statement> while (<condition>);

# while Loop

- ▶ while (<condition>) <statement>
  - ▶ If the condition is initially false, the statement is never executed.
- ▶ do <statement> while (<condition>);
  - ▶ The statement is executed at least one.

# for Loop

```
for (<exp1>; <exp2>; <exp3>) <statement>
```

```
exp1;  
while (exp2)  
{  
    statement  
    exp3;  
}
```

```
for (i = 0; i < n; ++i)  
{  
    // do something  
}
```

# Infinite Loop

```
while (1)
{
    // do something
}
```

```
for (;;)
{
    // do something
}
```

# Infinite Loop

```
while (1)
{
    // do something
}
```

```
for (;;)
{
    // do something
}
```

Both are okay, but **for** may lead to fewer machine code on some platform, which means it is slightly more efficient.



# break and continue

## break

```
int n = 10;
while (1)
{
    if (!n)
    {
        break;
    }
    --n;
}
```

## continue

```
int i;
for (i = 0; i < 10; ++i)
{
    if (i == 0)
    {
        continue;
    }
    printf("%d\n", i);
}
```