CSE 1310 - Introduction to Computers & Programming

Introduction to C

Alex Dillhoff

University of Texas at Arlington

What is C?

- Developed in the 1970s by Dennis Ritchie at Bell Labs
- Designed for efficient translation into hardware instructions
  - Useful for embedded systems, critical systems, etc.
  - Not ideal for applications such as web development
- C is an *Imperative Language*
  - Statements are explicitly written
  - Contrast with *declarative*, which declares the overall logic of a system
  - Example: Haskell

What is C?

C is a *compiled* language

- Compiled
  - Entire source code translated to machine code
  - More correspondence with machine language
  - Example: C, Java
- Interpreted
  - Instructions executed as they are read
  - Generally slower execution time
  - Example: Python, JavaScript

Why C?

- High level language, but low compared to other high level languages
- Introduces concepts useful for future courses
  - Memory management
  - Pointers

The First Program

Write a program that **reads** two *integers* from the user, **adds** them together, and **prints** the result.

First, plan your solution using pseudocode!

The First Program

1. Read two integers from the user.
2. Add the integers together.
3. Print the resulting sum to the user.

The First Program

1. Read two integers from the user.
2. Add the integers together.
3. Print the resulting sum to the user.

Does this seem obvious?

The First Program

If it seems obvious then you are ready to start implementing.

If the solution is not clear, break down the steps further until it is obvious.

Onto a solution...

The First Program

```c
#include <stdio.h>

int main(void) {
    int a = 0;
    int b = 0;

    // Read values from command line.
    scanf("%d", &a);
    scanf("%d", &b);

    // Add the numbers.
    int result = a + b;

    // Print the result.
    printf("%d + %d = %d\n", a, b, result);

    return 0;
}
```

Preprocessor Directives

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

- Start with #
- Includes code from external files
- The compiler injects the code into the program
- Standard header files, such as `stdio.h`, are kept in a special location

Preprocessor Directives

```
#define PI 3.14
```

- Defines a constant value named `PI` with value 3.14
- Useful for establishing program-wide standards
  - Example: `#define VERSION 0.8`
  - Prevents us from *hard coding* values

Comments

```
// This is a comment.
/* This is also a comment. */
```

Comments are used to enhance readability and understanding of code. They are ignored by the compiler.

The *main* Function

```c
int main(void) {
    return 0;
}
```

- `main()` is a function (think $f(x)$)
- All program execution starts with `main()`
- Every C program *must* include it
  - Defined by ISO/IEC

The *main* Function

```
int main(void) {
    return 0;
}
```

- Return type `int`
  - Defines what *type* is returned
- `return 0;`
  - Must return the *type* specified by the function
  - Returns control to the calling function
  - `int main()` is a special case – returns control to the OS

Types

In C, each value has a *type*. The basic data types are

- `int` – Integer values (-2, 0, 5, etc.)
- `float` – Floating point values (-4.3, 0.1, 2.5, etc.)
- `char` – ASCII character values ('a', '4', '*', '?', etc.)

Variables

```
float a = 6.2;
```

- *Declared* with a **type** and **identifier**
  - **Type:** `float`
  - **Identifier:** `a`
- *Defined* with a value
  - **Value:** `6.2`

Variables – Identifiers

- Consist of letters, numbers, and underscores
- Cannot start with a number
- Cannot conflict with **reserved words**
  - Words that a part of the language itself
  - `main`, `int`, `struct`, `float`

Variables

Valid variables

```
int myVal = 0;
float float_ = 2.3;
char letter3 = 'a';
```

Invalid variables

```
int 1int = 0;
float float = 2.3;
char my-char = 'a';
```

ASCII Character Set

- A character set defines which characters can be used for source code
- Typically, C implementations use ASCII
- Possible to use others, but portability suffers
- `https://www.rapidtables.com/code/text/ascii-table.html`

Operators

```
int result = a + b;
```

- Operators perform some basic function, such as addition or subtraction.
- In C, operands should be of the same type.
- Operators work with mixed types, but may have unexpected results (more on this later).

Output

```
printf("%d + %d = %d\n", a, b, result);
```

- `printf` is called with 4 arguments
  - `"%d + %d = %d\n"`
  - `a`
  - `b`
  - `result`
- Control string `"%d + %d = %d\n"`
  - Contains instructions which define the output
  - Embedded conversion specifiers `"%d"`
  - Escape character `"\n"`
- Additional parameters
  - Must correlate to the number of embedded conversion specifiers
  - Should match the type of the embedded conversion specifiers

Running the Code

- C code needs to be **compiled**, which creates a binary executable
- `gcc code.c` produces a file named `a.out`
- You can specify the output file name with `-o`
- `gcc code.c -o program`

Running the Code

In the shell, a program can be run by writing out the path:

- `./a.out`
- `./` - Start in current directory
- `a.out` - Name of the default program

Compilation

- Code is translated into a program through compilation
- Compilation for C goes through 5 steps
  - Preprocessing
  - Translation
  - Optimization
  - Assembling
  - Linking

Preprocessing

- Removes any comments from the source file
- Expands macros (more on that later)
- Expands included header files
- Injects definition values

Translating

- Translates the output of preprocessing into assembly language
- The assembly language is dependent on the platform
  - Intel
  - ARM

Translating - Example

In C

```c
int main(void) {
    return 0;
}
```

In ARM Assembly

```
main:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        mov     r3, #0
        mov     r0, r3
        add     sp, fp, #0
        ldr     fp, [sp], #4
        bx      lr
```

Optimization

- Inlining - Removes function call overhead
- Loop Unrolling
  - Removes instructions that control the loop
  - Removes end of loop tests
  - Reduces branch penalties
  - Increases the binary size

Assembling

- Converts assembly code in machine code (binary)
- This is also known as object code
- If multiple files are present, multiple objects are created

Linking

- Merges object code from multiple objects
- Links library functions
- Static linking
  - Library code is explicitly copied in the result
- Dynamic linking
  - Name of the library is placed in binary and referenced

Input and Output with Variables

- Useful to read input from the keyboard or other programs
- Output to other programs or the shell also important
- Two functions from `<stdio.h>`
  - `int printf(const char *format, ...);`
  - `int scanf(const char *format, ...);`

Output with printf()

```
int printf(const char *format, ...);
```

- The first argument is the control string
- Subsequent arguments are values corresponding to *format specifiers*

Format Specifiers

- `%d` - Integer
- `%f` - Floating Point
- `%c` - Character
- `%p` - Address
- `%o` - Octal
- `%x` - Hexadecimal

Output – Examples

```c
#include <stdio.h>

int main(void) {
    // Declare variable `a`
    // and assign a value of 1
    int a = 1;
    printf("a: %d\n", a);
}
```

Output – Examples

```
#include <stdio.h>

int main(void) {
    // Declare variable `a`
    // and assign a value of 1
    int a = 1;
    printf("a: %d\n", a);
}
```

Output: a: 1

Output – Examples

```c
#include <stdio.h>

int main(void) {
    // Declare variable `a`
    // with no assignment
    int a;
    printf("a: %d\n", a);
}
```

Output – Examples

```c
#include <stdio.h>

int main(void) {
    // Declare variable `a`
    // with no assignment
    int a;
    printf("a: %d\n", a);
}
```

Output: ?

Local variables that are uninitialized will have an indeterminant value.

Output – Examples

```c
#include <stdio.h>

int main(void) {
    float a;
    printf("a: %f\n", a);
}
```

Output – Examples

```c
#include <stdio.h>

int main(void) {
    float a;
    printf("a: %f\n", a);
}
```

Output: ?

Output – Examples

```c
#include <stdio.h>

int main(void) {
    char a;
    printf("a: %c\n", a);
}
```

Output – Examples

```c
#include <stdio.h>

int main(void) {
    char a;
    printf("a: %c\n", a);
}
```

Output: ?

Output – Examples

```
#include <stdio.h>

int main(void) {
    char a;
    printf("a: %d\n", a);
}
```

Output – Examples

```c
#include <stdio.h>

int main(void) {
    char a;
    printf("a: %d\n", a);
}
```

Output: ?

Output – Examples

```c
#include <stdio.h>

int main(void) {
    int a = 1;
    int b = a + 1;
    printf("a: %d\n", a);
    printf("b: %d\n", b);
}
```

Output – Examples

```c
#include <stdio.h>

int main(void) {
    int a = 1;
    int b = a + 1;
    printf("a: %d\n", a);
    printf("b: %d\n", b);
}
```

Output

```
a: 1
b: 2
```

Output - Examples

```c
#include <stdio.h>

int main(void) {
    int a = 1;
    printf("%p\n", &a);
}
```

Output - Examples

```c
#include <stdio.h>

int main(void) {
    int a = 1;
    printf("%p\n", &a);
}
```

Output

```
0x7ffdb851d7d4
```

Output – Examples

Example Scenarios

- Using a single format specifier
- Using multiple specifiers
- Using the incorrect specifier
- Additional formatting options

Example code will be posted online

Sub-Specifiers

Additional sub-specifiers can give fine-grained control over output:

−	Left-justify within fixed width.
+	Force + before positive value.
#	Write decimal even if no decimal values are present.
0	Left-pad with zeros instead of spaces for fixed width output.

Sub-Specifiers

Width and precision share similar modifications:

`%[number]specifier`	Minimum number of characters to be printed.
`%*specifier`	Specify width as an additional argument (before formatted argument).
`%[.number]specifier`	Different output dependent on specifier. See documentation.
`%.*specifier`	Precision specified by argument before formatted argument.

Sub-Specifiers

Example: `sub_specifiers.c`

Printing to stderr

- printf() prints output to stdout
- What if we want to print to stderr
- Use the function fprintf()

Printing to stderr

```
int fprintf(FILE *f, const char *s, ...)
```

- `stderr`, `stdout`, `stdin` are *files*
- `fprintf(stderr, "Test Error");`
- `fprintf(stdout, "Same as printf");`

Input with scanf()

```
int scanf(const char *format, ...);
```

- Control string accepts multiple specifiers
- Return value is the number of values successfully read

Input – Examples

```c
#include <stdio.h>

int main(void) {
    int a;
    scanf("%d", &a);
    printf("%d\n", a);
}
```

Output

10

10

Input - Examples

Additional Format Options

```
float f = 1.3;
printf("%.2f\n", f);
```

Input - Examples

Additional Format Options

```
float f = 1.3;
printf("%10.2f\n", f);
```

Input – Examples

Example Scenarios

- Reading a single input
- Reading multiple input
- Adding junk to the control string
- Using the incorrect specifier
- Checking the return value

Example code will be posted online

Arithmetic Operators

C has several binary arithmetic operators

- $+$ addition
- $-$ subtraction
- $*$ multiplication
- $/$ division
- % remainder
- $=$ assignment

Arithmetic Operators

There are *unary* operators as well.

- $-$ negative
- $+$ positive

Arithmetic Operators

Many operators can be used together as long as there exists values to operate on.

- `3 - 2 + 1 / 5 * 6;`
- `float c_squared = a * a + b * b;`
- `float c = a * (a + b) * b;`
- `int negative = -1;`

Operators – Examples

- Order of operations
- Mixing types
- Including unary operators

Example code will be posted online

Compound Assignments

Shorthand assignments perform common operations.

- `+=`
- `-=`
- `/=`
- `*=`
- `%=`

Compound Assignments

The following all compute the same thing:

```
a = a + 1;
a += 1;
a++;
```

`++` and `--` are commonly called *increment* and *decrement* assignments, respectively.

Increment and Decrement

A note about *increment* and *decrement*: the placement of this operator affects the result.

```
int a = 1;
printf("%d\n", --a);
a = 1;
printf("%d\n", a--);
```

output

```
0
1
```

Example – Evaluation Order

What should the output of the following be?

```
int i = 1;
printf("%d%d%d\n", --i, i--, i);
```

Example – Evaluation Order

What should the output of the following be?

```
int i = 1;
printf("%d%d%d\n", --i, i--, i);
```

Output (maybe)

```
0 1 1
```

C99 Standard: Evaluation Order

C99 does not specify the order of evaluation. This is determined by the **compiler** so that it may optimize.

C99 Standard 6.5.2.2

The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence points before the actual call.

C99 Standard: Evaluation Order

C99 Standard 6.5

Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.

```c
int i = 1;
printf("%d%d%d\n", --i, i--, i);
```

In this example there are 3 modifications between the first sequence point (function call) and the subsequent ;