Lecture P1: Introduction to C

#include <stdio.h>
int main(void) {
  printf("This is a C program\n");
  return 0;
}

Learning C

No prior programming experience assumed.
- Although it will make things easier.

Programming is learned with practice.
- Don’t expect to learn solely from these lectures.
- Do exercises.
- Experiment with code on your own.

Do reading.
- K&R for people with programming experience.
- Deitel & Deitel for beginners.
  - first 170 pages first two weeks
  - next 100 pages third week

C Background

Born along with Unix in the early 1970’s.
- One of most popular languages today.

Features.
- Exposes much of machine detail.
  - remember abstractions?
  - C exposes low-level abstractions
- Concise language.

Consequences.
- Positive: you can do whatever you want.
  - flexible and powerful
- Negative: you can do whatever you want.
  - shoot yourself in the foot

Aspects of Learning to Program

C Syntax
- Learning English.

Algorithms
- Learning to tell a coherent story (not necessarily in English).

Libraries
- Learning to reuse plots written by others.

These are different skills and learning processes.
An Example

Print a table of values of function $f(x) = 2 - x^3$. A first attempt:

```c
#include <stdio.h>

int main(void) {
    float x, y;
    printf("x     f(x)\n");
    x = 0.0;
    y = 2.0 - x*x*x;
    printf("%4.1f %6.3f\n", x, y);
    x = 0.1;
    y = 2.0 - x*x*x;
    printf("%4.1f %6.3f\n", x, y);
    "...
    x = 1.9;
    y = 2.0 - x*x*x;
    printf("%4.1f %6.3f\n", x, y);
    return 0;
}
```

Printf Library Function

Contact between your C program and outside world.
- Puts characters on “standard output.”
- By default, stdout is the “terminal” that you’re typing at.

Internally, all numbers and characters represented in BINARY (0’s, 1’s).
- `printf` converts from binary to more useful form (int, float).

Formatted output.
- How do you want the numbers to look?
  - integers, how many digits?
  - real numbers, how many digits after decimal place?
- Very flexible, see K&R pp. 13, 154.

Anatomy of Printf

float x, y;
x = 0.927;
y = 2.2;
printf("\n is newline character
%4.1f %6.3f\n", x, y);

Running a Program in Unix

When you type commands, you are controlling an abstract machine called the “Unix shell.”

- Compile: convert the program from human’s language (C) to machine’s language (stay tuned).
  - 1st try: syntax errors in C program
  - eventually, a file named a.out
- Execute: start the machine (at first instruction corresponding to first statement of `main`).
  - 1st try: semantic errors in C program
  - eventually, desired “printf” output
Anatomy of a While Loop

Previous program repeats the same code over and over.

- Repetitive code boring to write and hard to debug.
- Use while loop to repeat code.

```
while (condition) {
    statements;
}
```

C code

```c
x = 0.0;
while (x < 2.0) {
y = 2 - x*x*x;
    printf("%f %f", x, y);
    x = x + 0.1;
}
```

Table:

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>2.000</td>
</tr>
<tr>
<td>0.1</td>
<td>1.901</td>
</tr>
<tr>
<td>0.2</td>
<td>1.810</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

While Loop Example

Print a table of values of function f(x) = 2 - x³. A second attempt.

```c
#include <stdio.h>

int main(void) {
    float x, y;
    printf(" x     f(x)\n");
    x = 0.0;
    while (x < 2.0) {
        y = 2.0 - x*x*x;
        printf("%4.1f %6.3f\n", x, y);
        x = x + 0.1;
    }
    return 0;
}
```

Anatomy of a For Loop

The for loop is another common repetition structure.

```
for (expr1; expr2; expr3) {
    statements;
}
```

```c
for (x = 0.0; x < 2.0; x = x + 0.1) {
y = 2 - x*x*x;
    printf("%f %f", x, y);
}
```
For Loop Example

Print a table of values of function $f(x) = 2 - x^3$. A third attempt.

```c
#include <stdio.h>

int main(void) {
    float x, y;
    printf(" x     f(x)\n");
    for (x = 0.0; x < 2.0; x = x + 0.1) {
        y = f(x);
        printf("%.1f %6.3f\n", x, y);
    }
    return 0;
}
```

table3.c

uses for loop

Anatomy of a Function

Convenient to break up programs into smaller modules or functions.

- Layers of abstraction.
- Makes code easier to understand.
- Makes code easier to debug.
- Makes code easier to change later on.

$1.2 \quad f(x) = 2 - x^3 \quad 0.272$

float f(float x) {
    return 2 - x*x*x;
}

function in C

Anatomy of a Function

C function similar to mathematical function.

- Prototype or interface is first line of C function.
  - specifies input argument(s) and their types
    - can be integers, real numbers, strings, vectors, user-defined
  - specifies return value

Body or implementation.

- The rest, enclosed by { }

```c
float sum(float x, float y) {
    float z;
    z = x + y;
    return z;
}
```

sum function

output type    function name

scratch space
statements
stop execution
of function

input 2
type
input 2
name

output value

Function Example

Print a table of values of function $f(x) = 2 - x^3$. A fourth attempt.

```c
#include <stdio.h>

float f(float x) {
    return 2.0 - x*x*x;
}

int main(void) {
    float x;
    printf(" x     f(x)\n");
    for (x = 0.0; x < 2.0; x += 0.1) {
        printf("%.1f %6.3f\n", x, f(x));
    }
    return 0;
}
```

table4.c

no need for { } if only one statement

x += 0.1 is shorthand in C for $x = x + 0.1$
What is a C Program?

C PROGRAM: a sequence of FUNCTIONS that manipulate data.
- **main** function is first one executed.

A FUNCTION consists of a sequence of DECLARATIONS followed by a sequence of STATEMENTS.
- Can be built-in like `printf`.
- Or user-defined like `f` or `sum`.

A DECLARATION names variables and defines type.
- `float` `float x;`
- `integer` `int i;`

A STATEMENT manipulate data or controls execution.
- **assignment:** `x = 0.0;`
- **control:** `while (x < 2.0) {...}
  x = x + 0.1;`
- **function call:** `printf(...);`

Anatomy of a C Program

```c
#include <stdio.h>
float f (float x) {
  return 2.0 - x*x*x;
}
int main() {
  float x;
  x = 0.0;
  while (x < 2.0) {
    printf("%4.1f %6.3f\n", x, f(x));
    x = x + 0.1;
  }
  return 0;
}
```

Random Integers

Print 10 “random” integers.
- Library function `rand()` in `stdlib.h` returns integer between 0 and `RAND_MAX - 1` (usually 32767).

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int i;
  for (i = 0; i < 10; i++)
    printf("%d\n", rand());
  return 0;
}
```

Random Integers

Print 10 “random” integers between 0 and 599.
- No precise match in library.
- Try to leverage what’s there to accomplish what you want.

```c
#include <stdio.h>
#include <stdlib.h>
#define N 600
int randomInteger(int n) {
  return rand() % n;
}
int main(void) {
  int i;
  for (i = 0; i < 10; i++)
    printf("%d\n", randomInteger(N));
  return 0;
}
```
**Random Real Numbers**

Print 10 “random” real numbers between 0.0 and 1.0.
- No precise match in library.
- Try to leverage what’s there to accomplish what you want.

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

int main(void) {
    int i;
    for (i = 0; i < 10; i++)
        printf("%f\n", 1.0 * rand() / RAND_MAX);
    return 0;
}
```

Received output:

0.513871
0.175726
0.308634
0.534532
0.947630
0.171728
0.702231
0.226417
0.494766
0.124699

**Unix**

```bash
% gcc real.c
% a.out
```

---

**Random M x N Pattern**

Top-down design.
- Break a big problem into smaller subproblems.
- Break down subproblems into sub-subproblems.
- Repeat until all details filled in.

```c
#include <stdio.h>
#define M 9
#define N 9

int randomInteger(int n) {...}

int main(void) {
    int i, j;
    for (i = 0; i < M; j++) {
        for (j = 0; j < N; j++) {
            if (randomInteger(2) == 1) printf("*");
            else printf(" ");
        }
        printf("\n");
    }
    return 0;
}
```

**Print a random element.**
**Print a random row.**
**Print a random M x N pattern.**
Libraries

How is library function `printf()` created?
- User doesn’t need to know details (see COS 217).
- User doesn’t want to know details (abstraction).

How is library function `rand()` created?
- Linear feedback shift register? Cosmic rays?
- Depends on compiler and operating system.
- Caveat: “random” numbers are not really random.
  - can never have all properties of random bits
  - computers do exactly what we tell them to do!
- Note: on many systems, our `randomInteger` is very bad.

Moral: check assumptions about library function.

Gambler’s Ruin

Simulate gambler placing $1 even bets.
- Will gambler always go broke.
- If so, how long will it take if gambler starts with $c?

Gambler’s Ruin

Simulate gambler placing $1 even bets.
Q. How long does the game last if we start with $c?

```c
int main(void) {
    int i, cash, seed;
    scanf("%d %d", &cash, &seed);
    srand(seed);
    while (cash > 0) {
        if (randomInteger(2) == 1)
            cash++;
        else
            cash--;
        for (i = 0; i < cash; i++)
            printf(" ");
        printf("*
");
    }
    return 0;
}
```

Unix

```
% gcc gambler.c
% a.out              % a.out
4 1231  4 1234
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *
*  *  *  *  *

Hmmm.
```
Top-Down Design of Numerical Experiment

Goal: run an experiment to determine how long does it take to go broke.
- Find out how this changes for different values of c.

For all initial cash values between 2 and 9
run numerical experiments

repeat 5 times
how long before ruin?

Do gambler’s ruin and return value

Gambler’s Ruin Experiment

single experiment (code as before)

repeat for all initial
cash values 2 to 9

repeat 5 times

Gambler’s Ruin Experiment

Unix

<table>
<thead>
<tr>
<th>initial cash</th>
<th># bets</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>243</td>
</tr>
<tr>
<td>6</td>
<td>494</td>
</tr>
<tr>
<td>7</td>
<td>299</td>
</tr>
<tr>
<td>8</td>
<td>218</td>
</tr>
<tr>
<td>9</td>
<td>174090315</td>
</tr>
</tbody>
</table>

How long will it take to go broke?

Layers of abstraction.
- Random bit → gambler’s ruin sequence → experiment.

Programming Advice

Understand your program.
- What would the machine do?

Read, understand, and borrow from similar code.

Develop programs incrementally.
- Test each piece separately before continuing.
- Plan multiple lab sessions.
Debugging

Find the FIRST bug and fix it.

Syntax error - illegal C program.
  ■ Compiler error messages are good - tell you what you need to change.

Semantic error - wrong C program.
  ■ Use “printf” method.

Always a logical explanation.

Enjoy the satisfaction of a fully functional program!

Programming Style

Concise programs are the norm in C.

Your goal: write READABLE and EFFICIENT programs.
  ■ Use consistent indenting.
    – automatic indenting in emacs
  ■ Choose descriptive variable names.
  ■ Use comments as needed.

“Pick a style that suits you, then use it consistently.”

-Kernighan and Ritchie

Summary

Lots of material.

C is a structured programming language.
  ■ Function, while loop, for loop.
  ■ Can design large robust programs with these simple tools.

Programming maturity comes with practice.
  ■ Everything seems simpler in lecture and textbooks.
  ■ Always more difficult when you do it yourself!
  ■ Learn main ideas from lecture, learn to program by writing code.