Memory & Compilation

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

January 30, 2023

Lesson Plan

To keep in sync with the other sections we'll do some review (ρ) from last week's slides, which were updated.

1. Memory, Arrays, Pointers (ρ)
2. Functions (ρ)
3. Strings as Arrays
4. Compilation
5. Build Automation (Makefiles)

Acknowledgements

Part of the slides are borrowed from Karen Reid and Andi Bergen.

Section 1

Memory & Arrays (review)

Memory (ρ)

- The operating system manages the real memory based on hardware.
- From our perspective we're working with virtual memory on top.
- Bytes are typically the smallest unit of memory.
  - Each unit has an address, which is an integer-like numeric that can be operated on with integers.
- The address of a variable/struct is the address of its first byte.
- Local-scope variables are typically allocated memory on the *stack*.
- Dynamic allocation to the *heap* is explicitly handled (seen later).

Arrays: Declaration & Allocation (ρ)

Arrays are sequence of uniformly-sized elements stored in a contiguous region of memory. They're declared to contain types (such as characters or numerics) and an array size between brackets `[size]`.

```
float A[65];
A[0] = 6.0;
A[1] = 3.141592654;
printf("%f\n", A[1]); // bad approx of pi
printf("%f\n", A[2]); // undefined
```

A normal declaration assigns a region of memory to the array, but doesn't normally re-initialize the values.

Static Arrays

However, `static` arrays do initialize values as 0.

```
static long B[4];
printf("%ld\n", B[3]); // 0
```

Arrays: Initialization (ρ)

Alternatively, you can directly initialize arrays with values.

- Values in the array beyond the initializer are initialized as 0.

```
int csc209[4] = {2, 0, 9}; // csc209[3] == 0
```

- Due to type inference, the size of such declarations is optional.

```
int csc369[] = {3, 6, 9}; // size inferred
```

Arrays: Bounds in Memory (ρ)

- C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined.
- No run-time check of array bounds: behaviour exceeding bounds is undefined. If lucky, it might (appear to) work with no side effects.
  - Sometimes it'll do something random, harmless or not.
  - Worst-case, it might crash the program or OS.

```
int csc469 = {2, 2, 0, 8};
csc469[4] = 1; // will likely crash with stack smashing
```

- **Warning:** It is the programmer's responsibility to keep track of the size of an array! Take care not to violate the bounds of the array.

Arrays: Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/arraysVarLength.c`

Pointers: Arithmetic (ρ)

Pointers are technically numbers, so you can add integers to them. Then, you can access other values with relative pointers.

- If `p` points to `A[i]`, other `A[j]` can be accessed by performing arithmetic on `p`.
- C supports exactly these three forms of pointer arithmetic:
  - pointer + integer; pointer - integer; or pointer - pointer
- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to. That is, if `p` points to the array element `A[i]`, then `p+j` points to `A[i+j]`.
  - In other words, `A + i` is the same as `&A[i]` because both represent a pointer to element `i` of `A`.
  - Similarly, `*(A+i)` is equivalent to `A[i]` because both represent `i`'th element of `A`.
  - Assuming 32-bit integers, each increment on a pointer will move 4 bytes down, giving us the pointer to the next element.

Pointers: Arithmetic (Memory Addresses) (ρ)

```c
#include <stdio.h>
int main () {
    int A[] = {1, 2, 4, 8, 16, 32, 64};
    for (int i=0; i<6; i++)
        printf("A[%d]: addr %x; val %d\n", i, &A[i], A[i]);
    return 0;
}
```

Note the 4-byte intervals of consecutive addresses in contiguous memory.

```
A[0]: addr f4ec9730; val 1
A[1]: addr f4ec9734; val 2
A[2]: addr f4ec9738; val 4
A[3]: addr f4ec973c; val 8
A[4]: addr f4ec9740; val 16
A[5]: addr f4ec9744; val 32
```

Pointers: Arithmetic (Memory Addresses) (ρ)

```
#include <stdio.h>
int main () {
    int A[] = {1, 2, 4, 8, 16, 32, 64};
    for (int i=0; i<6; i++)
        printf("A[%d]: addr %x; val %d\n", i, &A[i], A[i]);
    return 0;
}
```

Another run... the addresses (or rather, base addresses) always change, depends on memory.

```
A[0]: addr b49d4720; val 1
A[1]: addr b49d4724; val 2
A[2]: addr b49d4728; val 4
A[3]: addr b49d472c; val 8
A[4]: addr b49d4730; val 16
A[5]: addr b49d4734; val 32
```

Pointers: Arnold's Examples

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/crazyPointers.c`

`mcs.utm.utoronto.ca/~209/23s/lectures/src/c/pointersAndFunctions.c`

Section 2

Functions (review)

Functions: Arguments by Value (ρ)

C passes arguments by value. Implicit casting is performed on numerical function arguments; beware of truncation!

```
#include "math.h"
#include "stdio.h"
int as_long(long l) { return l; }
float as_float(float d) { return d; }
int main() {
    int nine_plus_ten = 21;
    long massive = __LONG_MAX__ - nine_plus_ten;
    printf("%ld -> %d\n", massive, as_int(massive));

    double pi = M_PI;  // approximate the approximation
    printf("%1.32f -> %1.32f\n", pi / 2, as_float(pi) / 2);
    return 0;
}
```

Full code

github.com/rhubarbwu/csc209/blob/master/lectures/lec04/arg_cast.c

Functions: Arguments by Value (ρ)

What does this do to mass? It's being passed by value.

```
#include <stdio.h>
#define half_life 12
#define time 100

void decay(double mass) {
    mass /= 2;
}
int main() {
    double mass = 244817;
    for (int i = 1; i < time; i++)
        if (i % half_life == 1)
            decay(mass);
    printf("After %d, %lf remains.\n", time, mass);
    return 0;
}
```

Functions: We need pointers! (ρ)

- Easy access to and abstraction of complex structures.
- Allows reference to the same data when desired.
- Pointers consume less memory than deep copies.
- Convenient null values for initialization/error-checking.

Pointer Arguments

Pointers allow you to pass primitives or structures by reference, rather than value. Instead of copying and passing the entire structure, copy/pass the pointer(s) in constant time.

Functions: Arguments by Reference (ρ)

What about this? It's passed by reference.

```c
#include <stdio.h>
#define half_life 12
#define time 100

void approx_decay(double *mass_ptr) {
    *mass_ptr /= 1.4142857;
}
int main() {
    double mass = 244817;
    for (int i = 1; i < time; i++)
        if (i % (half_life / 2) == 1)
            approx_decay(&mass);
    printf("After %d, %lf remains.\n", time, mass);
    return 0;
}
```

Because pointers can point to anything, you also have pointers to pointers.

```
int main() {
    int i = 81; int *pt = &i; int **pt_ptr = &pt;

    int *r= *pt_ptr; // intermediate dereference
    int k = *r; // complete the dereference

    int k1 = **pt_ptr; // direct double dereference

    int ***pt_ptr_ptr = &pt_ptr; // triple pointer
    int k2 = ***pt_ptr_ptr;
    return 0;
}
```

Source: PCRS (University of Toronto)

Functions: Pointers & Arrays (ρ)

The relationship between pointers and arrays in C is a close one. Understanding this relationship is critical for mastering C.

- C allows to perform addition and subtraction on pointers to array elements. This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.
- Pointers can point to array elements. Here's an example:

```c
int a[10], *p;
p = &a[0];
*p = 5; // stores 4 in a[0]
```

- A pointer is not an array but it can contain the address of an array. An array is not a pointer either but the compiler interprets the name of an array as the address of its 0th element.

```c
int *x = &a[0];
int *y = a;
```

Functions: Arrays as Arguments/Parameters (ρ)

When passed to a function, an array name is treated as a pointer. That is, what is passed to the function decays to a pointer to the first element.

```
int find_largest(int a[], int n){
    int i, max = a[0];
    for (i = 1; i < n; i++)
        if (a[i] > max) max = a[i];
    return max;
}
int main() {
    ...
    find_largest(A, N); // A is not copied;
    ...                 // rather, a points to A[0]
    return 0;
}
```

Functions: Arrays as Arguments/Parameters (ρ)

The size of an array is not inherently stored in the array itself; the only way to know/pass on how large the array is is to pass the length of the array alongside.

- Remember `argv`? It's an array of "strings" of length `argc`.

```
int main(int argc, char **argv) { return 0; }
```

Strings

"Strings" are actually `char`-arrays, i.e. `char *`. They are *null-terminated*: their last values are the `\0` to indicate the end of the string; more about this when we discuss strings...

Functions: Arrays as Arguments/Parameters (ρ)

1. An array used as an argument isn't protected against change.
2. Just like with all structures, latency and bandwidth of passing an array are not affected by the size of the array.
3. An array parameter can be declared as a pointer if desired.
   - Although declaring a parameter to be an array is the same as declaring it to be a pointer, the same isn't true for a variable.

   ```c
   int A[10]; // allocates memory for 10 integers
   int *a;    // allocates memory for a pointer, not array
   ```
4. A function with an array parameter can be passed an array "slice":

   ```c
   find_largest(&b[5], 10);
   ```

Functions: Arnold's Examples

mcs.utm.utoronto.ca/~209/23s/lectures/src/c/functions/functions.c

Section 3

Strings: Just Spicy Arrays

Strings: Just Spicy Arrays

"Strings" in C are actually a special case of `char` arrays: they're **null-terminated**, meaning the last *actual* character is `\0`.

```
char limited[9]; // such a string shouldn't exceed 8
```

- When working with strings, `\0` isn't typically used/printed.
- Instead, it indicates where the string ends.
  - If you're writing a function that doesn't know the exact length of the string, the `\0` might come in handy.
  - Inserting a `\0` in the middle of a `char *` shortens the effective string.

    ```
    char *city = "mississauga";
    city[4] = '\0'; // city is now "miss"
    ```
- Declaring strings with explicit length initializes remainder as `\0`.
- Important for many string-wise functions.

Strings: Delicate Arrays

String manipulation in practice (if not carefully done) often results in unexpected/inconsistent behaviour or memory/pointer errors.

```c
int main() {
    char utm_local[] = "erindale"
         utm_city[] = "mississauga",
         utsg_local[] = "st. george",
         utsg_city[] = "toronto";
    utm_city[4] = '\0';  // utm_city is now "miss"

    utm_local[8] = 'i'; // what happens to utm_local?
    utsg_local[10] = 'k'; // what about now?
}
```

Full code

github.com/rhubarbwu/csc209/blob/master/lectures/lec04/campuses.c

Strings: Delicate Arrays

Depending on stack memory layout and changes, removing \0 might lead to something "harmless" like inconsistent reading overruns.

```
#include "strings.h" // using strlen
int main() {
    char F[6] = "{'a', 'p', 'p', 'l', 'e'}", // F[5] = \0
        P[6] = "nachos",  // no space for \0
        M[6] = "popcorn"; // n is truncated, no \0
    printf("%d %d %d\n", len(F), len(P), len(M));

    char B[9] = "sourdough";  // try len = 5 or 13
    printf("%d %d %d\n", len(F), len(P), len(M));
    return 0;
}
```

Full code

github.com/rhubarbwu/csc209/blob/master/lectures/lec04/foods.c

Section 4

Compilation

Compilation: C Programs

- C programs can consist of multiple `*.c` files
- Each individual `*.c` file can be compiled to an object file.
- Object files (`*.o`) contain "placeholders" for addresses of functions that were declared but not defined.
  - Header (`*.h`) files ensure consistency between function declarations across your program's multiple source files.
- The linker connects object files together to create an executable file.

Compilation: GCC Pipeline (ρ)

Recall that all input files (the last arguments) flow through the pipeline (depending on options) up to (and including)...

1. Preprocessing (`gcc -E`) strips comments and expands directives.
2. Compilation (`gcc -S`) generates assembly code (`*.s`/`*.asm`).
3. Assembly (`gcc -c`) generates binary/machine code objects (`*.o`).
4. Linking (`gcc`) consolidates objects into a single application.
    - defaults to `a.out`, but you can use `-o <output>` to specify.

Compilation: Preprocessing

Preprocessing...

- removes comments.
- expands compiler directives
  - `#includes` statements (akin to imports).
  - `#define` macros.

You can emit preprocessed code with `gcc -E`.

```
$ gcc -E foods.c
$ gcc -dM -E foods.c # includes pre-defined macros
```

Seems like a lot of steps, right? It's like building a large project bottom-up with the ability to mix/match components when necessary/desired. An analogy could be assembling automobiles.

Automobile Manufacturing	Compiling with GCC
Extract raw materials	Preprocess compiler directives
Produce basic parts	Compile to assembly
Assemble larger parts like the engine	Assemble to binary objects
Assemble together, connect pipes/wires, screw/plug everything else, etc.	Link to an application

At any point, different parts can be chosen/substituted. The interface just has to be valid (recall CSC207) as per the header files (`*.h`).

Compilation: Separate Compilation

Suppose you want the following English code… (`en.c`)

```c
#include <stdio.h>
#define fmt "Hi, %s. My name is %s too!\n"
char *name = "Peter", *my_name = "Erika";
int salutation() {
    printf(fmt, name, my_name);
    return 5;
}
int main() {
    printf("%s: %d\n", name, salutation());
    return 0;
}
```

Notice the `#include` imports and macro `fmt`.

Compilation: Separate Compilation

But you want to make a French version too... (`fr.c`)

```
#include <stdio.h>
#include <string.h>
char* name = "Pierre";
int salutation() {
    printf("Bonjour, %s.\n", name);
    return strlen(name);
}
int main() {
    printf("%s: %d\n", name, salutation());
    return 0;
}
```

You could write separate programs, but what if they're largely similar? Can we flexibly reuse consistent code?

Yes! Tie them together with a common header file (such as `lang.h`).

```
#include <stdio.h>
#include <string.h>
extern char* name; // global declaration
void salutation();
```

And use it as an interface by `#include` directive.

```
#include "lang.h"
int main() {  // perform a greeting
    salutation();
    printf("%s: %d\n", name, strlen(name));
    return 0;
}
```

Notice `salutation()` now returns `void` instead of `int`. Why? How?

Here's what French (`fr.c`) might look like.

```
#include "lang.h"
char* name = "Pierre";
void salutation() {
    printf("Bonjour, %s.\n", name);
}
```

And English (`en.c`) ...

```
#include "lang.h"
#define fmt "Hi, %s. My name is %s too!\n"
char *name = "Peter", *my_name = "Erika";
void salutation() {
    printf(fmt, name, my_name);
}
```

Finally, compile for the language you want. Here the objects are compiled separately for clarity and reuse; you can link complete binaries without recompiling the components each time.

```
gcc -S en.c              # compile en.s
gcc -c en.s              # assemble en.o
gcc -o en en.o greet.c   # link English binary
./en                     # run English binary

gcc -c en.c fr.c         # create en.o and fr.o
gcc -o fr fr.o greet.c   # link French binary
./fr                     # run French binary
```

Compilation: Arnold's Examples

mcs.utm.utoronto.ca/~209/23s/lectures/src/c/logistics.zip

Section 5

Build Automation (Makefiles)

Automation with Scripts

Now, suppose we are writing a language translation program that uses an intermediate representation (IR) of type `int *` of length 2048.

- Input from the source language is *encoded* to the IR.
- Output to the target language is *decoded* from the IR.
- Don't worry about the implementation of `encode()` and `decode()`.
- Suppose for every language `xx`, we have `xx-e.c`…

  ```
  int *encode(char *input);
  ```

  And `xx-d.c`…

  ```
  char *decode(int *ir);
  ```

- And that `main()` could call either of these at will.

How would you design and build this?

Automation with Scripts

One solution is to put `encode()` and `decode()` in a common header(s) and compile each language pair manually.

```
gcc -o ar-bn ar-e.c bn-d.c main.c
gcc -o ar-de ar-e.c de-d.c main.c
gcc -o ar-en ar-e.c en-d.c main.c
gcc -o ar-es ar-e.c es-d.c main.c
gcc -o ar-fr ar-e.c fr-d.c main.c
gcc -o ar-hi ar-e.c hi-d.c main.c
gcc -o ar-jp ar-e.c jp-d.c main.c
gcc -o ar-pt ar-e.c pt-d.c main.c
gcc -o ar-ru ar-e.c ru-d.c main.c
gcc -o ar-zh ar-e.c zh-d.c main.c
...
```

We might want to write a script/function to generalize...

```
$ gcc -o $1-$2 $1-e.c $2-d.c.
```

You could even batch it...

```
#!/bin/sh
langs="ar bn de en es fr hi jp pt ru zh"
for l1 in $langs; do
    for l2 in $langs; do
        gcc -o $1-$2 $l1-e.c $l2-e.c
    done
done
```

This is much better, right? More elegant and programmable for sure.

You could even batch it...

```
#!/bin/sh
langs="ar bn de en es fr hi jp pt ru zh"
for l1 in $langs; do
    for l2 in $langs; do
        gcc -o $1-$2 $l1-e.c $l2-e.c
    done
done
```

Is this efficient? Definitely not! Every time you need some encoder `xx-e` or decoder `yy-d`, you're recompiling the same object but not reusing it.

Makefiles: Introduction

- Makefiles facilitate building (i.e., compiling, linking, sometimes testing and packaging) projects consisting of multiple source files.
- If only one source file has changed, no need to recompile everything; instead:
  1. Recompile source files that have changed.
  2. Relink updated object files to generate new executable file.

Makefiles: Format

A Makefile contains a sequence of rules, each in the format:

```
target: prereq_1 prereq_2 ... prereq_n
    action_1
    ...
    action_n
```

Makefiles are processed by the make program

- Run `make` with no arguments to evaluate first rule.
- Run `make TARGET` to execute action(s) defined in rule for `TARGET`.
    - Only if `TARGET` prerequisites were modified since last time that `make TARGET` was run.
- To force `make TARGET` to recompile code, you can:
    - Update last modified time of prerequisite source files, with `touch`; or
    - Delete prerequisite object files.

Makefile Syntax: Defining Variables

You may define variables; e.g., to store compiler flags:

```
CFLAGS= -g -Wall -Werror -fsanitize=address

reverse : reverse.c
    gcc $(CFLAGS) -o reverse reverse.c
```

You can even declare an alternative compiler.

```
CXX=clang++

forward : forward.c
    gcc $(CFLAGS) -o forward forward.c
```

Makefile Syntax: Automatic (Built-In) Variables

Variable	Meaning
`$@`	Target
`$<`	First prerequisite
`$?`	All out of date prerequisites
`$^`	All prerequisites

```
CFLAGS= -g -Wall -Werror -fsanitize=address

hello: hello.c hello.h
    gcc $(CFLAGS) -o $@ $<
```

Ref.: 10.5.3: Automatic Variables, GNU Make manual

Makefile Example: Past Assignment

```
FLAGS= -Wall -Werror -fsanitize=address -g
OBJ = simfs.o initfs.o printfs.o simfs_ops.o
DEPENDENCIES = simfs.h simfstypes.h

all : simfs

simfs : ${OBJ}
    gcc ${FLAGS} -o $@ $^

%.o : %.c ${DEPENDENCIES}
    gcc ${FLAGS} -c $<

clean :
    rm -f *.o simfs
```

Makefile Example: Pattern Rules

```
%.o : %.c ${DEPENDENCIES}
    gcc ${FLAGS} -c $<
```

- Most files are compiled in the same way, so we write a pattern rule for the general case
- % expands to the stem of the file name (i.e., without extension)
- `gcc -c` compiles the source file(s), but does not link

Makefile Example: Phony Targets

You may want a command that builds a target:

```
OBJ = simfs.o initfs.o printfs.o simfs_ops.o

simfs: ${OBJ}
    gcc ${FLAGS} -o $@ $^
```

Or a target that doesn't build anything:

```
clean:
    rm -f *.o simfs
```

Section 6

Makefiles: Practice

Makefiles: Practice

Provided by Karen Reid from my CSC209H1 in Fall 2019, supposedly an assignment from an even older offering of CSC209.

Makefiles: Practice (1)

```
test_print: test_print.o ptree.o
    gcc -Wall -g -std=gnu99 -o test_print test_print.o ptree.o
```

1. What's the target?
2. What're the prerequisites? What's another term for them?
3. How many actions does this rule have?
4. What does a file that ends in `.o` contain? How is it generated?

Makefiles: Practice (2)

```
FLAGS = -Wall -g -std=gnu99
DEPENDENCIES = ptree.h
all: test_print print_ptree
test_print: test_print.o ptree.o
    gcc ${FLAGS} -o $@ $^
print_ptree: print_ptree.o ptree.o
    gcc ${FLAGS} -o $@ $^
%.o: %.c ${DEPENDENCIES}
    gcc ${FLAGS} -c $<
clean:
    rm -f *.o test_print print_ptree
```

1. If we were to run `make print_ptree` which rule is evaluated first?
2. What new files would be created?
3. What is the last action that is executed in the make command above?

Makefiles: Practice (3)

```
FLAGS = -Wall -g -std=gnu99
DEPENDENCIES = ptree.h
all: test_print print_ptree
test_print: test_print.o ptree.o
    gcc ${FLAGS} -o $@ $^
print_ptree: print_ptree.o ptree.o
    gcc ${FLAGS} -o $@ $^
%.o: %.c ${DEPENDENCIES}
    gcc ${FLAGS} -c $<
clean:
    rm -f *.o test_print print_ptree
```

4. Which files will the pattern rule (`%.o : %.c`) match on?
5. If we the modify `ptree.c` and run `make print_ptree` again, which rules are evaluated? Which actions are executed?