CSC161 2010F *Imperative Problem Solving*

Class 06: Types and Operators

Held: Monday, 6 September 2010

Summary: We continue our explorations of the basic design of C, focusing on types in C.

Related Pages:

- EBoard.
- Reading: K&R 2 ; GNU Coding Standards, 5.1-5.4.

Notes:

- I expect to respond to Assignment 1 this evening, but it may take until tomorrow evening.
- Are there questions on Assignment 2?
- For tomorrow, read sections 7.1-7.4 of K&R.
- Many of the examples from today's class can be found in Examples/Compiling in the course web.

Overview:

- Questions and Answers.
- Hello World.
- Compiling C Programs.
- C Beyond K&R.
- Back to the Basics.
- Lab (if time permits).

Questions and Answers

- Are there particular questions on the reading? (I may just accumulate them and then answer them as best I can in the appropriate sections of class.)
- Optionally: Quiz!

Hello World

Let's explore how one turns the K&R "Hello World" program into something that meets the GNU coding standards.

```
#include <stdio.h>

main()
{
    printf("hello world\n");
}
```

Compiling C Files

- In our *nix system, the C compiler command has a form similar to that of many other applications.

  `cc` *`flags files`*

- The simplest instruction to compile a C file is

  `cc` *`file`*`.c`

- Provided this instruction succeeds, it creates an executable named `a.out`.
- Our C compiler is fairly friendly, so it will let a lot of stuff slip. You tell it to warn you about some dangerous stuff by adding the `-Wall` flag.

  `cc -Wall` *`file`*`.c`

- If you want even more information about what might be wrong with your program, use the C program checker called `splint`

  `splint` *`file`*`.c`

- Each of the `cc` instructions above calls the executable `a.out`. What if you want a different name? You use the `-o` *`outfile`* command line parameter.

  `cc -Wall -o` *`executable file.c`*

- The command “compile a C file and use the prefix of the C file as the name of the executable” is so common that there’s an automated way to do that.

  `make` *`file`*

- This command uses the program Make, which lets you set rules for building different files. Make has some decent defaults for C.
- You will notice that, by default, Make does not use the `-Wall` flag. You can tell it to do so by adding an appropriate flag to the command line.

  `make CFLAGS=-Wall` *`file.c`*

- But it’s painful to type all that extra stuff each time. Fortunately, we can put the flags in special file called `Makefile`.
  - The Makefile contains

    ```
    CFLAGS=-Wall
    ```

  - The command can once again be

    `make` *`file`*

- I strongly recommend that you make it a practice to use Make to build programs.
  - We’ll look at the details of Make throughout this semester.

C Beyond K&R

- Although K&R do not mention it, it's good style to return a value from `main`.
  - Why? Becuase `main` is declared to return a value.
- The return value indicates to the caller (typically the operating system) whether your program exited normally (return 0) or abnormally (anything else)
- What codes should you return upon failure?
  - When I started programming, there were no standards.
  - These days, there are two competing standards
    - Two choices: `EXIT_SUCCESS` or `EXIT_FAILURE`
    - Some folks use `<sysexits.h>`
    - You can read the latter file with

      ```
      less /usr/lib/sysexits.h
      ```

- You can also use `exit (CODE)` as the last line of `main`.
  - You need to include `stdlib.h` to use `exit`.

Back to the Basics

- Let's move on to Chapter 2 of K&R. As you can tell, this chapter gives us a lot of language basics, focusing on the basic types of the language.

Variable Names

- Variable names are composed of letters, numbers, and underscores.
  - In Scheme, we got to use many other symbols.
- Depending on the C compiler you use, not all of the characters you use are significant.
  - Our compiler seems to differentiate variable names even to the 1000th character.

Declarations

- Simple form:

  ```
  type name;
  ```

- In C, declarations need to come at the start of a block.
  - In a variety of C-like languages, declarations can come in many other places.

Assignment:

- Standard form

  ```
  variable = expression;
  ```

- Meaning:
  - Evaluate the expression
  - Copy the value into the memory location associated with the variable

Primary Types

- C has only a few basic types: `char`, `int`, `short`, `long`, `float`, and `double`, along with a few variations of those types.
  - Signed or unsigned.
- Other types are usually represented by these types or combinations thereof.
  - For example, Boolean values are represented as integers. (0 is false, anything else is true.)
- All of these types use a fixed number of bits.
  - So the programmer is expected to understand representations
  - And we therefore need different kinds of integers.
  - You can find information on limits in `<limits.h>`
  - We'll look in detail about representations in a few days.
- Typically, C is silent about overflow.
  - If you add two numbers and the number is too large to represent, you just get some other number, rather than an error message.
- There are way too many ways to represent numeric constants
- For convenience, we'll focus on `char`, `int`, and `double`

Enums

- We can create our own "enumerated" types: Types whose values appear to be symbolic.
  - Behind the scenes, they are just integers.
- Basic syntax for describing the type

```
enum name { VAL0, VAL1, ..., VALN };
```

- Basic syntax for declaring enumerated variables

```
enum name var;
```

And More

- We'll cover the other stuff as necessary.

Lab

- Do the lab.
- Be prepared to reflect.
