More C for Java Programmers

CS152

Chris Pollett

Sep. 10, 2008.

Outline

- Struct’s
- Memory allocation
- File I/O
- Buffering
- make

Struct's

- C has a mechanism for collecting together a bunch of existing data types into a new one using struct's.
- These can be thought of as classes without member functions.

```
struct Person
{
    char name[12];
    int age;
}; //notice the ;
struct Person p, *ptr;
strcpy(p.name, "Bob"); //in string.h. Note p.name[4] = '\0' after
// many useful string functions like strlen, strcmp, etc are in string.h
p.age = 5;
ptr = &p;
printf("%d %s %s", p.age, (*ptr).name, ptr->name);
```

More on struct's

- You can also declare:

```
struct
{
        int a,b;
} test;
test.a = 5; /* so have declared a variable test but have not given the kind of struct it is a name */
```

- The syntax struct Person p; of the last slide is sometimes awkward. To simplify it you can write

```
typedef struct Person person_type;
person_type a,b,c;
```

- Using structs and pointers you can create recursive data structures:

```
struct mylist
{
        int a;
      struct mylist *next, *prev;
} test;
```

- Remark: can fake classes by using struct's which have function pointers as members.

Memory Allocation

- Sometimes we don't know how much memory we need for a job at compile time. For instance, we might not know how big a list will grow or how big an array we need to hold a string.
- C has a runtime heap and supports allocating/deallocating memory from it at runtime:

```
#include <stdio.h>
#include <stdlib.h> //for malloc and free
int main()
{
    int *p;
    p = (int *)malloc(10*sizeof(int)); /*sizeof returns number of bytes an int takes (could do
     sizeof(person_type)) from last slide */
   if( p == NULL)
   {
       return 1; //bail out
    }
    /* do stuff. To refer to the location of ith int can do (p + i), its value is *(p + i) or p[i]
   */
   free(p); // got to free or create a memory leak -- unlike Java no garbage collection
   return 0;
}
```

- Notice we cast the result of malloc to be of type int rather than void*.

File I/O

- The usual C File I/O is tightly connected to the Unix notion of a stream.
- We have already been using streams: name printf sends its data to the default output stream, stdout.
- Functions for I/O are all mainly in stdio.h. To see what are available functions look in Wikipedia.
- There is a also a stdin and and a stderr. For example,

```
int a;
scanf("%d", &a); //reads from stdin one int into a.
```

- Functions like printf, scanf, getc, etc which operate on the standard streams all have analogs which operate on files: fprintf, fscanf, fgetc, etc.

Example Reading From a File in C

```
#include <stdio.h>
int main(int argc, char * argv[]) //notice getting command-line args
{
  int c;
  FILE *fp;
  if(argc < 2)
  {
     return 1; //bail if no file specified

  }
  fp = fopen(argv[1], “r”); // r is for reading, w for write, rb for binary, etc
  while ((c = fgetc(fp)) != EOF)
  {
     printf(“%c”, (char)c );
   }
   fclose(fp);
   return 0;
}
```

Buffering

- As an example of how the language and the platform are connected, consider input in C on Unix:

```
printf("Hit a key to continue");
c= getchar();
```

- Although we are only requesting a single character from stdin, since in Unix stdin is line-buffered, we have to wait till someone hits enter to get our character.
- In Dos C which has a different default buffering, we wouldn't have to hit enter.
- We can make OS calls to change the buffering in Unix, but this just shows, how we program is influenced by the platform we are on:

```
#include <termios.h>
//…
   struct termios tio;
   tcgetattr( 0, &tio );
   tio.c_lflag &= ~ICANON;
   tcsetattr( 0, TCSANOW, &tio );
   printf("Hit a key to continue");
   c= getchar();
```

make

- make is a build utility -- a utility to compile and link large software projects -- developed by Stuart Feldman at Bell Labs in 1977.
- It heavily influenced many later build tools such as ant, and it also has been ported to many platforms. For example, Microsoft OS's use nmake.
- make is very similar in some ways to Prolog, and can be viewed as perhaps the most commonly used declarative language.
- Typically make is run from the command-line with a line like:

  make *target*
- The make utility would then search the current directory for a file called Makefile and then tries to satisfy the target goal.

Makefile Structure

- A Makefile consists of rules of the form:

```
target1: depends_on1 depends_on2 …
<tab>command1
<tab>command2
…
<blankline>
target2: depends_on1 depends_on2 … #etc
```

- # is used for a single-line comment
- Notice the use of tabs is important!
- Here are some example targets:

```
myprog: myprog.o
	cc -o $@ $<

myprog.o: myprog.c
	cc -c -o $@ $<
# $@ refers to the target $< refers to the first dependency
clean:
	rm -f myprog myprog.o
```

More on Makefiles

- You can declare variables in a Makefile using the format varname = value like:

```
CC = gcc
SUBDIRS = io linkedlist
```

- These variables could then be used:

```
all : $(SUBDIRS)
	$(CC) historylesson.c -o historylesson
```

- An example of a multi-line make rule might be something like:

```
io :
	@echo "Making io..."
	cd io
	make all
```

- Make uses the file modification dates to figure out what needs to be re-compiled. Typically, it only performs incremental compiles.
- There are various shortcuts you can use for rules that I won't go into very much. For example, if one had the target hello.o . It would match the rule:

```
%.o : %.c
```