CS246—Selected C Review Topics

Spring 2016

Declaration and Definition

Declaration: Asserts enough about the existence of an entity to permit type-checking to proceed; no further details.

Definition: Provides full details about the entity, and causes space to be set aside for it (in the case of variables and functions). Note that definition does **not** imply initialization.

Examples:

Entity	Definition	Declaration
variable	`int x;`	`extern int x;`
function	`int f(int x){ . . . }`	`int f(int x);`
structure	`struct S { . . . };`	`struct S;`

An entity can be declared any number of times, but must be defined at most once.

Control structures

Conditional:

```
if (condition) {
  stmts
}
else if (condition) {
  stmts
}
...
else {
  stmts
}
```

Loops:

```
while (condition) {
  stmts
}

for (initialization; condition; update) {
  stmts
}
```

`break` exits the current loop

`continue` skips the rest of the current loop iteration and starts the next loop iteration

Pointers and Arrays

```
int *x;  // x is a pointer to an int; x holds the address of an int

int y[] = {2,4,6,8,10};  // x is an array of ints
```

The name of an array is shorthand for a pointer to the array's first element.

But arrays and pointers are **not** the same.

On the other hand, when an array is passed as a function parameter, a pointer to the array's first item is what is actually passed. Therefore, a function header

```
int f(int a[]) { ... }
```

is identical in meaning to

```
int f(int *a) { ... }
```

Constants

```
const int x = 5;   // x is a constant

struct S { int x, y; };

const struct S s;  // s is a constant structure; its fields cannot be changed

const int *p = &n;  // p is a non-constant pointer to a constant int
                    // *p cannot be assigned (e.g., *p = 5;  NO)
                    // p can be reassigned (e.g., p = &m;  OK)
```

Note that, although we cannot modify `n` by assigning to `*p`, we may still be able to modify `n` directly, if `n` itself was not declared `const`.

```
int * const p = &n;  // p is a constant pointer to a non-constant int
                     // *p can be reassigned (e.g., *p = 5;  OK)
                     // p cannot be reassigned (e.g., p = &m;  NO)
```

It may help to read C declarations starting from the variable and working outwards, in order to get their meaning right.

```
const int * const p = &n;  // p is a constant pointer to a constant int
                           // Neither p nor *p may be reassigned.
```

Strings

```
char msg[] = "Hello!";
```

Strings in C are null-terminated arrays of characters. The string `msg` above is an array consisting of 7 characters: the H, e, l, l, o, ! above, plus a null character at the end of the string to indicate where it ends.

Careful:

```
char a[] = "Hello!";
char *b = "Good-bye!";
```

The array `a` resides on the stack. The pointer `b` points to the beginning of a sequence of characters. That sequence is stored in static memory (not on the stack), and typically it resides in read-only memory. Attempting to modify the contents of `b` will likely crash the program:

```
a[1] = 'u';  // No problem
b[2] = 'a';  // Will probably crash
```

Command-line arguments

```
int main(int argc, char *argv[]) {
  . . .
}
```

- `argc` is the number of command-line arguments, and is always at least 1;
- `argv` is an array of strings, each one denoting one command-line argument;
- `argv[0]` is the name of the program itself, as it was typed on the command line (which is why `argc` is always at least 1);
- actual command-line arguments are `argv[1]`, ..., `argv[argc-1]`;
- `argv[argc]` is guaranteed to be a null pointer.