U N I V E R S I T Y

CS 240: Programming in C

Lecture 18: Types
Type Qualifiers
Storage Classes
The C Preprocessor

Prof. Jeff Turkstra

Announcements

- Midterm Exam 2 approaching
  - Thursday, April 10 8pm – 10pm
  - New seating chart

Takehome Quiz 8

- Take your career account username and repeat it three times. E.g.:
  - loginloginlogin, turkstraturkstraturkstra
- One character at a time, insert the first 9 characters into a binary tree. For loginloginlogin, the root would be “l”, then you would insert “o”, then “g”, etc
  - The left child should be less than or equal to its parent.
- Draw the resulting tree with a circle for each tree node and the character inside
- For comparisons, the ordering of characters is 0-9a-z:
  - 0123456789abcdefghijklmnopqrstuvwxyz

Types

- We are already familiar with the many basic types in the C language. They are called first-class types:

void	nothing. (1 byte?)
char	usually 1 byte
short	usually 2 bytes
int	usually 4 bytes
long	usually 8 bytes; sometimes 4 bytes
long long	usually 8 bytes (gcc only); even on 32-bit systems
float	usually 4 bytes
double	usually 8 bytes

Sizes of data

- There are general rules-of-thumb for the size of a variable, depending on type. The only way to be sure is to use sizeof().
- There are rules that say, for instance, that an int must be no smaller than a short and no larger than a long
- Types are automatically promoted to the next larger type of the same family (e.g., integer or floating point) within arithmetic operations

Conversion

- After promotion, arguments to an operator are checked
  - If the same, proceed
  - Otherwise conversions may take place
  - For each type, if one of the arguments is that type, the other is converted to the same type:
    - long double, double, float, unsigned long, long, unsigned, int

Type modifiers

- Integer types (char, short, int, long, long long) can have an additional modifier to indicate to the compiler whether the datum represents a signed or unsigned (always positive) value. E.g.:

```
unsigned char x = 200;  /* OK */
signed char y = 200;    /* overflow… */
```

- For non-integer types, “signed” has no meaning
- The default modifier for int is signed
- What’s the default modifier for char?

Second-class types

- Constructed types are second-class types. They are created by the programmer.
- Examples of derived types include anything that the programmer declares that is a struct, union, enum, or pointer to anything

Assignments

- You can make assignments between compatible types or types that can be promoted. This usually works between all first-class types. E.g.:

```
int i;
unsigned int ui;
float f;
char c;
…
f = ui;
i = f;
c = f;
```

Bad assignments

- You cannot make assignments between data of differing second-class types:

```
struct my_struct {
  int x;
} str1;

struct your_struct {
  int x;
} str2;

str2 = str1;  /* not allowed */
```

str1 and str2 are equivalent but differ by their type name.

Type qualifiers

- There are two type qualifiers that can be used with any type declaration:
  const: this datum must not be modified
  volatile: this datum may be modified by something outside the program! (e.g. the hardware, another program, multi-threaded programs)
- Only one at a time may be used for any single declaration

Type qualifier examples

```
const double PI = 3.14159265358979323846;

int main() {
  const int factor = 45;

  return my_sub(factor);
}
```

const pointers

- The const keyword can be used with pointer declarations in interesting ways:
  `const int *ptr;`
  - Means that ptr points to an integer whose value cannot be modified

  `int * const ptr;`
  - Means that ptr is an unmodifiable pointer that points to an integer whose value *can* be changed
- How about: `const int * const ptr;`

const pointer arguments

- Here is the actual prototype for strcpy():
  `char *strcpy(char *dest, const char *src);`
  - This is a guarantee made by the author of strcpy() that the string passed through the src argument will not be modified
  - The string that is passed through src does not need to be defined as const
- Anytime you create a function whose arguments accept a pointer whose dereferenced values will not be modified, those arguments should be declared as const

const pointer examples

```
unsigned int count_tree_nodes(
  const struct tree_node *root) {
  if (root == NULL)
    return 0;
  return 1 + count_tree_nodes(root->left)
           + count_tree_nodes(root->right);
```

```
unsigned int strlen(const char *str) {
  unsigned int len = 0;
  while (*str++ != '\0')
    len++;
  return len;
}
```

Are we modifying anything that str points to here???

Purdue Trivia

- Elliott Hall of Music was dedicated on May 3 and 4, 1940 with more than 11,000 people attending
  - Included a recital by opera stars Helen Jepson and Nino Martini
- Seats 6,005 on three levels – one of the largest proscenium theaters in the world
- Named after Edward C. Elliott, president of Purdue 1922-1945
- "Sister" to Radio City Music Hall
  - J. Andre Fouilhoux, designer of New York's Radio City Music Hall, served as one of Elliott Hall's architects along with Walter Scholer
  - 5 seats larger
- John Johnson, an artist from Frankfort, IN created the sculptures

Storage classes

- There are two storage classes in the C language that we really care about. (There are two more, but you'll rarely type them!)
  - extern: The datum is defined in some other module
  - static: (If the datum is a local variable) the datum is initialized only once and retains its value between invocations of the function
  - static: (If the datum is a global variable) the datum is not visible from other modules
- Note: use either extern or static, but not both for a particular datum

When to use extern...

- If you are developing an application that has global variables whose values are accessed from multiple C files, each variable must only be defined in one C file and defined as extern in all other modules where the variable is referenced

Two modules

- module1.c:

```
unsigned int counter;
double temp;
```

- module2.c:

```
extern unsigned int counter;
extern double temp;

void increment_count() {
  counter++;
}
```

static local variable

- Consider a function that we want to use to generate and return a new serial number each time it's called...

```
unsigned int new_serial() {
  static unsigned int serial = 45000;
  return serial++;
}

int main() {
  printf("First: %d\n", new_serial());
  printf("Second: %d\n", new_serial());
  return 0;
}
```

What will the two printf()s print?

static global variables

- Use a static global when the variable must be visible to other functions in the same module but must not be seen (or called) from other modules that are linked into the application...

```
static int private_data;
void my_function() {
  private_data = 15;
}

static void increment_private() {
  private_data++;
}
```

Why do we use any of these?

- You can get away with writing any program without any type qualifiers or storage classes (except extern)
- Using static improves software modularity and makes you less prone to violate an assumption that you may have made long ago (or many lines ago)
- Using const reminds you (or guarantees to a customer) that something should not be modified
- If you actually need to use volatile you'll usually know why... ;-)

The preprocessor

- When a .c file is compiled, it is first scanned and modified by a preprocessor before being handed to the real compiler
- If the preprocessor finds a line that begins with a #, it hides it from the compiler and makes special note of it
  - Or, perhaps, takes other actions
- We've seen only two preprocessor directives so far:
  - #define and #include

#include

- #include always pulls a header file into another file
  - #include “file.h”
    Pull in file.h from the present directory
  - #include <file.h>
    Pull in /usr/include/file.h

Example of #include

/home/jeff/x.c

```
#include <stdio.h>
#include "x.h"

int main() {
  printf ("Val %d\n", X);
  return 0;
}
```

/usr/include/stdio.h

```
/*
 * scary things
 * in this file...
 */
typedef FILE ...
```

/home/jeff/x.h

```
#define X  ( 3456 )
```

Final result of #include

```
/*
 * scary things
 * in this file...
 */
typedef FILE ...

#define X  ( 3456 )

int main()
{
  printf ("Val %d\n", X);
  return 0;
}
```

- All of the things that previously resided in separate files were pulled together into one stream
- This gets fed to the compiler

More preprocessor directives

- This might be best done by example:

```
#define TESTING

  x = some_function(y);
#ifdef TESTING
  printf(“Debug point!\n”);
  x = x + 5;
#else
  x = x + 5;
#endif
```

- If we turn off the TESTING variable, the debug statements are no longer delivered to the compiler

More preprocessor directives

- This might be best done by example:

```
/* #define TESTING */

  x = some_function(y);
#ifdef TESTING
  printf(“Debug point!\n”);
  x = x + 5;
#else
  x = x + 5;
#endif
```

- If we turn off the TESTING variable, the debug statements are no longer delivered to the compiler

More preprocessor directives

- More flexible directives...

```
#if defined(TESTING) && !
defined(FAST)
  printf(“Debug point!\n”);
#endif
```

- You can have mathematical expressions also...

```
#define FLAG 46
#if (FLAG % 4 == 0) || (FLAG == 13)
...
#endif
```

You can #define macros...

- You can create something that looks like a function but just gets substituted at compile time:
  `#define INC(x)  x + 1`
  Notice, no semi-colon!
- So the following statement:
  `printf("I like the number %d\n", INC(z));`
  becomes, at compile time:
  `printf("I like the number %d\n", z + 1);`

Ternary operator

- Some C operators take one operand: &, *, -, ...
- Many C operators take two operands: +, /, %, ...
- One C operator takes three operands:
  `x = a ? b : c;`
  - This is the ternary operator. It means "if a is non-zero, then use the value b. Else, use the value c."
  - We typically use it in macros

More macros

- Find the absolute value:
  `#define ABS(x)   x < 0 ? -x : x`
- Find the highest number:
  `#define MAX(x, y)x > y ? x : y`
- Problems result if you say something like:
  `A = ABS(B + C);`
  `A = B + C < 0 ? - B + C : B + C;`
- So we add parentheses around the substitution variables to make them safe.

Safer macros

- Find the absolute value:
  `#define ABS(x) ( (x) < 0 ? -(x) : (x) )`
- Find the highest number:
  `#define MAX(x, y)  ( (x) > (y) ? (x) : (y) )`
- A longer one:

```
#define RET_ON_ERROR(x) \
    if ((x) < OK) { \
      printf("ERROR: %d\n", (x)); \
      return (x); } \
```

Why macros?

- Runtime efficiency
  - The preprocessor replaces the macro identifier with the token string.
  - No overhead of a function call.
- Passed arguments can be of any type. Why is this fact so cool??
  `#define MAX(x, y) ((x) > (y) ? (x) : (y))`
  - We only need one macro for finding the highest number regardless if the arguments were ints, floats, doubles, even chars. They all work.
    - A function called max() would not be this flexible

Other preprocessor tricks

- You could spend a lot of time looking at nuances of the preprocessor
- Consider the following:
  `printf("The date is %s\n", __DATE__);`
- Most of the preprocessor features are for advanced software development practices.
  - If you create a large software project in C, someone in your development team should be a preprocessor expert
- Read Chapter 4.11 for more info

For next lecture

- Read 4.11 (12.13 in Beej's)
- Topics for next time:
  - Callbacks
  - Optimizing code
  - More fun stuff!

Boiler Up!