ECE 131 – Programming Fundamentals

Module 2, Lecture 3: Data Types, Variables and Expressions – Variables & Constants in C

Dr. Daryl Lee

University of New Mexico

Variable and Constants

- Before a data type can be used in a program, the data type must be instantiated. I.e., an instance of that data type must be created.
- When a data type is instantiated, the compiler sets aside (i.e., allocates) the memory required by the data type.
- There is one definition of a given data type, but we can create many instances of that data type.
- An instance of a data type might be a constant, in which case the value it stores cannot be changed, or it might be a variable, in which case the value it stores can change (through an assignment) while the program is running.
  Ex: The value of $\pi = 3.1415926\ldots$ should be declared as a constant if you use it in a program, its value should not change! However, if your program uses a counter to keep track of something, you would want to make that counter a variable.

Type Declarations

- We have already seen the C syntax for declaring (i.e., instantiating) variables:

  Ex: `int num;`

  We say that "`num` is a variable of type `int`", and the entire statement is referred to as a type declaration.

- If the keyword `const` is placed before any type declaration, then the value that is stored cannot be changed during program execution.

  Ex: `const int num = 12;`

  In this case we say that "`num` is a constant of type `int`". Constants must be defined where they are declared.

- In both examples, `num` is referred to as an identifier.

Type Declarations

- More formally, the syntax for a type declaration in C consists of a valid C type followed by a valid C indentifier:

  <modifier> data_type identifier1 <, identifier2, . . . >;

  where the elements shown between "<" and ">" are optional.
- We have already seen one modifier, `const`, that may appear at the beginning of a type declaration.
- There are a number of other modifiers that may also appear here. We'll introduce them shortly.

Identifier Names

- The rules for identifiers in C:
  - An identifier may consist of a sequence of letters (upper- or lower-case), digits, and underscore characters; however, the first character may not be a digit.
  - An identifier may not be any of the reserved keywords in C. (See Table A.2 on pg. 426 of Kochan).

Ex:
```
int 2d_point, 3d_point;    // invalid
int int2;      // valid
float xyz$abc;   // invalid
const long double main, if;   // invalid
char MyChar_for_me;   // valid
```

Type Declarations and Initialization

Here's the complete syntax for declaring a C variable:

<modifier> data_type identifier < = initial_value>;

where there can be more than one identifier/inital_value pair.

- If an initial value is supplied, this value is stored in the memory that is allocated when that variable is created.

  Ex:
  ```
  long int num = 23, counter = 0;
  const double val = 10.67894;
  ```
- If you use the `const` modifier, then it's essential that you also assign an initial value. Why?
- The compiler attempts to match the initial value that is supplied to the data type.

Ex:
```
long int num = 23.2;   // will truncate to 23
```

Type Declarations and Initialization

- When initializing integer data types, an integer constant may be preceded by:
  - The digit "0". This indicates the the constant should be treated as an octal number.
    Ex: `int num = 0621;`
  - The characters "0x" or "0X". This indicates that the constant should be treated as an hexadecimal number.
    Ex: `int RGBcolor = 0xCCFF97;`
- For initializing floating-point data types, C supports a format based on scientific notation. In this case, the letter "e" (upper or lower case) is placed between the significand and the exponent.
  Ex: `float num = 1.292e9;`
  where `1.292e9` represents 1.292×10^9.
- For the `char` data type, a character constant is created by enclosing a character within single quotes.
  Ex: `char letter = 'a';`

Storage Class Specifiers

- In C, a storage class specifier may also appear as a modifier in a variable declaration.
- This modifier plays a part in determining the scope (or visibility) and persistence (or lifetime) of the variable.
- The scope of a variable determines what parts of a program are able to reference that variable.
- The persistence of an variable determines when the variable comes into existence (i.e., memory space is allocated for it), and when it is destroyed (i.e., the memory space it used is deallocated).
- The scope and persistence of a variable are determined by:
  - where it is declared, and
  - any storage class specifiers that are used in its declaration.
- The available storage class specifiers in C are: `auto`, `extern`, `register`, and `static`.

Automatic Variables

In C, an automatic variable is any variable that is declared inside a block.

- The scope of an automatic variable is local to the block that contains it.
- An automatic variable persists from the time it is declared, until execution reaches the end of the block that contains it.

  Ex:

  ```
  main()
  {
     auto float num;
  }
  ```

- By default, if no other storage class specifiers are used, the keyword `auto` is assumed for any variables declared inside a block. Thus, the keyword `auto` can be removed in the example above, and nothing would change.

THE UNIVERSITY of NEW MEXICO

Automatic Variables

- The name automatic variable comes from the fact that memory space for the variable is *automatically* allocated and de-allocated when program flow enters and leaves the variable's scope.
- Most of the variables we've declared so far are automatic variables.
- Indeed, this is the most common type of variable in typical C programs.
- Summary – An automatic variable has:
  - local scope,
  - temporary persistence (it exists only while the block that contains the variable is executing).
- For an automatic variable, scope = persistence.

External Variables

- External variables have global scope, and persist for the entire time the program is running.
- Any variable that is declared outside of a function is an external variable.

  Ex:

  ```
  float num;   // an external variable
  main()
  {
     statements
  }
  ```
- A local variable may have the same name as a global variable. References to that name in the scope of the local variable will access the local variable, not the global variable.

THE UNIVERSITY of NEW MEXICO

External Variables

- The keyword `extern` is needed if you want to specify that you want to use a global variable.
  Ex:

```
func()
{
   extern float num;    // use the global variable
}                       // num declared elsewhere
```

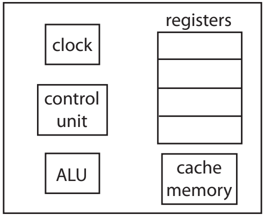
- If the keyword `extern` were left off of the variable declaration above, you would be declaring an automatic variable, completely separate (with its own memory) from a global variable called `num` declared at a global scope.
- When should you use global variables and constants?

Static Variables

- The keyword static may be placed before any variable declaration. Ex: `static float num;`
- Exactly where in a program the declaration occurs determines what affect the static keyword has.
- If the variable declaration creates a variable with local scope, then the static keyword changes the persistence of that variable only.
- The variable's scope will remain local, but it will persist for the entire time the program is running. I.e., scope $\subseteq$ persistence.
- If `static` is used while declaring a variable with global scope, the persistence of the variable is not changed, but the scope is modified.
- Specifically, the variable will now be known by all functions in the file in which the variable is declared, but it will not be known by functions that are part of the program but declared in a different file.

Register Variables

- The keyword register may be placed before any automatic variable declaration. Ex: `register float num;`
- This is a suggestion to the compiler that the variable or constant should be placed in a CPU register.



- Why do this? For performance reasons, you want the the variable to be accessed as quickly as possible.
- Note that this is just a suggestion — the compiler may choose to ignore it.
- Over the years, compilers have become very good at producing optimized code. Thus, this keyword is rarely necessary.

Storage Class Specifiers

```
main()
{
   extern int a;  // tell the compiler that a exists somewhere.
   float x;   // known only inside main().
   register int y;  // try to place y in a CPU register.
   func();
}

int a = 2;  // here a is actually declared.
              // a is known everywhere in the program
void func()
{
   static int x;  // known only inside func(), but not in block below.
                    // the value stored in x will be retained from one call
                    // to the next of func().
  {
     char x;   // known only inside this block in func()
     x = a;
  }
}
```