16.070 Introduction to Computers and Programming

1 March 2001            Recitation 4            Spring 2001

Topics

Data Types in C
- Fundamental Data Types
- ASCII Character Table
- Arithmetic Conversions
- Named Constants & Definitions

Data Types in C
By now you have seen many references to variables and the added complication that they can be of different types. C is known as a **Strongly Typed** language which always requires a user to specify what a piece of data is before using it. So why create a language this way? Well, for one thing, it allows a user to control the amount of space that variables take by trading off variable resolution (number of bits with which a variable is represented) with the amount of memory or disk space that is available. It also acts as a safeguard when a program is compiled, e.g. generating a compilation error when a variable of type float (floating point) is treated as being of type char (character). Sometimes compilers just warn us when this happens, indicating to the programmer that he/she should check if the code is doing what he/she would like it to do. Some languages are **Loosely Typed**, such as BASIC.

Lets look at a few examples:
This example creates a variable called `Age_of_Dog`, of type `int` (integer):

```c
int Age_of_Dog;
```

We could even load the age of the dog into the variable when it is defined by coding:

```c
int Age_of_Dog=5;
```

Or declare multiple variables in the same line such as:

```c
int Age_of_Dog, Age_of_Cat, Age_of_Ma_Baker;
```

We have also seen examples of how functions may accept as input variables of a specific type. The following code indicates that a function called `How_Old_Is_Ma` returns an integer value after accepting two integer variables:

```c
int How_Old_Is_Ma(int Birth_Year, int Present_Year);
```

In the next section we tabulate some differences between variables of different types.
Fundamental Data Types

The fact is that data types are platform specific. IBM, SGI, Sun and manufacturers all adhere to their own standards regarding the amount of memory assigned for different data types. ANSI-C does provide a minimum standard, indicating AT LEAST how many bytes should be assigned to each standard data type.

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Keyword</th>
<th>Bytes Required</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>char</td>
<td>1</td>
<td>-2^7 to 2^7 -1</td>
</tr>
<tr>
<td>Integer</td>
<td>int</td>
<td>2</td>
<td>-2^15 to 2^15 -1</td>
</tr>
<tr>
<td>Short Integer</td>
<td>short</td>
<td>2</td>
<td>-2^15 to 2^15 -1</td>
</tr>
<tr>
<td>Long Integer</td>
<td>long</td>
<td>4</td>
<td>-2^31 to 2^31 -1</td>
</tr>
<tr>
<td>Unsigned Character</td>
<td>unsigned char</td>
<td>1</td>
<td>0 to 2^8 -1</td>
</tr>
<tr>
<td>Unsigned Integer</td>
<td>unsigned int</td>
<td>2</td>
<td>0 to 2^16 -1</td>
</tr>
<tr>
<td>Unsigned Short Integer</td>
<td>unsigned short</td>
<td>2</td>
<td>0 to 2^16 -1</td>
</tr>
<tr>
<td>Unsigned Long Integer</td>
<td>unsigned long</td>
<td>4</td>
<td>0 to 2^32 -1</td>
</tr>
<tr>
<td>Single-Precision Floating-Point*</td>
<td>float</td>
<td>4</td>
<td>-10^38 to 10^38</td>
</tr>
<tr>
<td>Double-Precision Floating-Point**</td>
<td>double</td>
<td>8</td>
<td>-10^308 to 10^308</td>
</tr>
</tbody>
</table>

* Approximate precision to 7 digits
** Approximate precision to 19 digits

Table 1  ANSI-C minimum memory requirements for standard data types

How do we know how much memory is allotted for each data type on your specific platform? Programming languages have a `sizeof()`, or equivalent, statement. It is always good style to use `sizeof()` in C when you rely on the amount of memory used by your program. This statement returns the size of any data type e.g.

```c
int a;
a=sizeof(int);
printf("integers have size: %d bytes",a);
```

Output:

```
integers have size 4 bytes
```

Press any key to continue

You will notice that integers on the PC platform, running MS Windows, consume more memory than the minimum ANSI specification of 2 bytes.

ASCII Character Table

You may have noticed that the above table specifies a variable of type `char` to have a range from 0 to 255. How can a character than correspond to a number? Well,... The ASCII (“American Standard Code for Information Interchange”) codes are used to represent characters as one byte integers. The first 128 of them (0 → 127) are the standard ASCII characters, while the next 128 (128 → 255) are the extended ASCII characters (symbols, accented letters, Greek letters, etc...).

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 NUL SOH STX ETX EOT ENQ ACK BEL BS HT LF VT FF CR SO SI
1 DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM SUB ESC FS GS RS US
2 SP ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
3 @ A B C D E F G H I J K L M N O
4 ` P Q R S T U V W X Y Z [ \ ] ^ _
5 a b c d e f g h i j k l m n o
6 p q r s t u v w x y z { | } ~ DEL
```

Table 2  Hex-indexed ASCII table
Some examples:

<table>
<thead>
<tr>
<th>Character</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>32</td>
<td>0010 0000</td>
</tr>
<tr>
<td>!</td>
<td>33</td>
<td>0010 0001</td>
</tr>
<tr>
<td>&quot;</td>
<td>34</td>
<td>0010 0010</td>
</tr>
<tr>
<td>#</td>
<td>35</td>
<td>0010 0011</td>
</tr>
<tr>
<td>$</td>
<td>36</td>
<td>0010 0100</td>
</tr>
<tr>
<td>A</td>
<td>65</td>
<td>0100 0001</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
<td>0100 0010</td>
</tr>
</tbody>
</table>

Control Codes:

- **NUL** (null)
- **SOH** (start of heading)
- **STX** (start of text)
- **ETX** (end of text)
- **EOT** (end of transmission) - Not the same as ETB
- **ENQ** (enquiry)
- **ACK** (acknowledge)
- **BEL** (bell) - Caused teletype machines to ring a bell. Causes a beep in many common terminals and terminal emulation programs.
- **BS** (backspace) - Moves the cursor (or print head) move backwards (left) one space.
- **TAB** (horizontal tab) - Moves the cursor (or print head) right to the next tab stop. The spacing of tab stops is dependent on the output device, but is often either 8 or 10.
- **LF** (NL line feed, new line) - Moves the cursor (or print head) to a new line. On Unix systems, moves to a new line AND all the way to the left.
- **VT** (vertical tab)
- **FF** (form feed) - Advances paper to the top of the next page (if the output device is a printer).
- **CR** (carriage return) - Moves the cursor all the way to the left, but does not advance to the next line.
- **SO** (shift out) - Switches output device to alternate character set.
- **SI** (shift in) - Switches output device back to default character set.
- **DLE** (data link escape)
- **DC1** (device control 1)
- **DC2** (device control 2)
- **DC3** (device control 3)
- **DC4** (device control 4)
- **NAK** (negative acknowledge)
- **SYN** (synchronous idle)
- **ETB** (end of transmission block) - Not the same as EOT
- **CAN** (cancel)
- **EM** (end of medium)
- **SUB** (substitute)
- **ESC** (escape)
- **FS** (file separator)
- **GS** (group separator)
- **RS** (record separator)
- **US** (unit separator)
Arithmetic Conversions

Sometimes equations in your code will contain variables of different types, and all in the same equation. What should the type of the answer to such an equation be? Standard C compiler rules exist that deal with such cases:

- C always converts the value of the “lower” type to that of the “higher” type, and then performs the operation, producing a value of the “higher” type (see hierarchy table below).
- Characters and short ints are converted to int before operations are performed.
- You can use casts to force the conversion of one data type to another. The use of a cast overrides the standard rules for type conversions. The cast operator does not change the value stored in the variable.
- Casting is done by explicit type conversions immediately before an expression. Place the type in parentheses immediately in front of the variable you want to change.
- Unsigned property does not necessarily propagate from lower to higher. Do not mix unsigned and signed data types in operations; use casts. The reason is that the storage of unsigned types varies from computer to computer and hence if you do not use casts, you may end up with unwanted results.
- All decimal numbers without a declared type default to double.

### Hierarchy of Signed Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>long double</td>
<td>unsigned long</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>unsigned</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>unsigned short</td>
<td></td>
</tr>
<tr>
<td>long int</td>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above table indicates that the *long double* data type is at the top of the hierarchy, with *char* being at the bottom.

Examples:
1. int a;
   double b;
   c=a+b; /* c would need to be of type double (Rule 1)*/
2. int a;
   c=a+’k’; /* c would need to be of type integer (Rule 2)*/

### Type Casting

Sometimes we don’t like the standard rules of arithmetic conversion to be applied to us. In such cases we may decide to cast variables or results of arithmetic into specific types. Let’s look at an example:

```c
double Velocity, Time_Elapsed;
Distance = Velocity*Time_Elapsed;
```

We would usually need to define *Distance* as being of data type *double*. A compiler like Visual C would even compile the code if *Distance* was defined as *int*, but it would issue a warning message. This is not the norm for all compilers!

Other C compilers, such as Interactive C, which we will use when programming the Handy Boards, are more strict about types. Many compilers don’t even hold with the arithmetic conversion rules that you have learned. The safest bet is to make use of a process called *type casting*. The correct way to cast a variable of one data type into another would be:
/* variable declaration */
int Distance;
double Velocity=10.0;
double Time_Elapsed=100.0;
/* Type casting */
Distance =(int)(Velocity*Time_Elapsed);

Named Constants & Definitions
A named constant is a variable whose value is fixed and cannot be changed during the execution of the
program. The declaration of a constant begins with the keyword `const`, followed by the type specifier for
that constant, followed by the name and assignment of value for that constant. It is typical to use all capital
letters when naming constants.

    const type NAME = value;

Constants come in handy when you need to represent a value that is used repeatedly throughout a program
and could change at some point. For example, if you wrote a program to simulate a Mars Lander but then
wanted to see how it would perform on another planet you could change all the gravity values in the
equations by changing one constant.

Let's look at an example where constants are useful:

    /**************************************************************************
    * TJ
    * 16.070
    * Recitation 3
    * Example
    **************************************************************************/
    /
    /* This program determines the speed at which
    a rock will hit the surface of a planet with
    gravity acceleration constant=Grav
    (ignoring drag) and initial drop altitude =
    InitAltitude */
    /*
    It also determines the amount of energy gained
    by the rock with mass=Mass */
    #include <stdio.h>
    #include <math.h>     /* to use sqrt() */
    const double Grav = 9.81;     /* in m/s^2 */
    const double InitAltitude = 1000;   /* in m     */
    const double Mass = 10;    /* in kg    */
    int main(void) /* start main */
    {
        double velocity,energy;
        velocity=sqrt(2*Grav*InitAltitude); /* velocity eqn. */
        printf("Impact Velocity = %.2lf m/s.\n",velocity);
        energy=Mass*Grav*InitAltitude; /* potential energy eqn. */
        printf("Impact Energy = %.2lf J.\n\n",energy);
        return 0;
    } /* end main */

A user would not have to bother with changing the mathematical equations of the example in order to
adapt it to solve the same problem for different planet.

Hardware
By now you are familiar with using desktop computers. But what really goes on behind that keyboard and
monitor? Have you ever had a look inside? If you haven’t, then today is your chance to see the different
parts of a computer up close and personal. We will show you how different (and how similar) embedded and desktop computers are.