C Building Blocks

Overview

In this lesson we will

✓ Introduce a very high level view of a C program
✓ Study basic building blocks of C a program
✓ Begin to explore data types that are part of the C language

These are known as the intrinsic (or built-in) types,

Introducing a C Program

intro0.c

#include <stdio.h>
int main(void)
{
    printf("Bonjour C program, ca va?\n");
    printf("Hello, yes, things are well, I'm running, it's good exercise you know\n");

    return 0;
}

With that introduction

We write C programs to help us solve problems

For that matter almost any programs

In solving problems we

✓ Enter questions into program in form of data
✓ Magic happens
✓ Get answers back

We have following simple high-level diagram

Questions

How do we get questions into a program
How do we get answers back

As for magic – that just happens don’t ask
Getting Questions In

We get questions in in variety of ways
  USB, Firewire, WIFI, Mouse
  Keyboard, Touch pad, Scanner, Gesture recognition
  Network
These should be familiar devices
All are pretty standard inputs

In C call such a collection – *standard inputs*
  Abbreviate as
    Standard In
    *Stdin*

Standard In
  Part of earlier architecture diagram shown as *input*

**Example**

getchar0.c

```c
#include <stdio.h>
int main(void)
{
    char a;
    printf("please enter a character\n");
    a = getchar();
    return 0;
}
```

This code fragment will
  • Ask the user for some data
  • Get that data

When you start to run a program
  Operating system connects program’s input to keyboard
    Which is typically *stdin*
How this happens is more magic
    Will examine more later
    For now we go with magic
Typing on keyboard now goes directly into program

At this point we claim character entered into program
    Prove it – I don’t see any character

Getting Answers Out
    Like input we get answers out in variety of ways
    USB, Firewire, WIFI
    Display, Printer, Aural or Video
    Network
These should be familiar devices
All are pretty standard outputs

In C call such a collection – standard outputs
    Abbreviate as
    Standard Out
    Stdout

Standard Out
    Part of earlier architecture diagram shown as output

Example
    print0.c

    #include <stdio.h>
    int main(void)
    {
        char a;
        printf("please enter a character\n");

        a = getchar();
        printf("the character is %c\n", a);
        return 0;
    }
When you start to run a program
  Operating system connects program’s output to display
    Which is generally `stdout`

How this happens once again more magic
  Will examine more later
  For now we’re on a roll with magic

Our high-level diagram now becomes

**In and Out Revisited**

In previous discussion
  Used two built-in functions for performing *input* and *output* operations
    For *output* used `printf()`
    For *input* used `getchar()`

Let’s now examine in greater detail
  *Standard I/O library* contains rich set of functions
    For supporting input and output from our program

We’ll start with output

**Program Output**

`printf` function supports what is called *formatted* output

Full syntax for `printf` function given as

```
syntax
  int printf("formatString", variableList);
```

The *formatString*

- Optional
- Enclosed in quotes “ “
- Terminated with comma ‘,’
- Specifies how output is to be formatted and presented on display
- Is a text string containing format variables that are to be instantiated

From the values specified in `variableList`
variableList is comma separated list of variables

Value of each formatVariable in formatString

Corresponds to value of corresponding variable in variableList

More commonly used format variables given in accompanying table

Example

print1.c

#include <stdio.h>
int main(void)
{
    // declare, define, and initialize some working variables
    int x = 5;
    int y = 10;
    int z = 0;
    float a = 5.6;
    char b = ‘e’;

    // perform a simple calculation
    z = x + y;
    // print the results of the calculation to the display
    printf("the sum of x and y is %d\n", z);
    // print a float
    printf("the float is %f\n", a);
    // print a character
    printf("the character is %c\n", b);

    return 0;
}

The character sequence \n
Called new line

When encountered in format string

Prints a cartridge return line feed combination

Moves printing to succeeding line

- 5 of 21 -
Program Input

`scanf` function supports what is called *formatted* input

Full syntax for `scanf` function given as

```
syntax
    int scanf("controlString", variableList);
```

The *controlString*

- Specifies what kinds of input are to be read
  Assigned to each of variables specified in `variableList`
- `variableList` is comma separated list of variables
- Value of each `controlVariable` in `controlString`
  Corresponds to value of corresponding variable in `variableList`

Variables on input

White space separated
- Read one at a time
  Until new line encountered
  New line left in input buffer
  Must remove manually

Most commonly used format variables follow those in `printf` `formatString`

**Example**

`scanf0.c`

```c
#include <stdio.h>
int main(void)
{
    // declare, define, and initialize some working variables
    int x;
    float y;
    char z;

    // read an int
    scanf("%d", &x);
    // read a float
    scanf("%f", &y);
    // read a character
    scanf("%c", &z);
```
// print the int
printf("the int is %d\n", x);

// print the float
printf("the float is %f\n", y);

// print a character
printf("the character is %c\n", z);
return 0;
}

Fundamental Data - Expressing Magic

For magic to happen – need to speak computerese
Be able to form questions computer can understand

Need language for computer
Language we will use called C
Begin by looking at some of words in C language
Call these identifiers

Identifiers in C

Name of object in C language
Called an identifier
Gives us means to refer to entity in program
Such an entity may be a
• Variable
• Function

Although any combination of symbols
Could be used as identifier
C standard establishes some restrictions
As to what constitutes legal identifier

Let's look at these rules
• An identifier is case-sensitive
  This means
  main() and MAIN() are two different functions
• The first character of an identifier must be an alphabetic character or an underscore
Some compilers permit a $ 
  This is non-standard and should be avoided
Some vendors prefix their identifiers
  With one or maybe two underscores
  C standard has specific rules allowing vendors to do this
Consequently identifiers that begin with single or double underscore
  Should be avoided

Example
  t1 and temperature1
  Both legal identifiers
  Which conveys more information

Identifiers should be as descriptive as possible
  Makes your program
  Easier to read
  Less error prone
  Typically we begin identifier with lower case letter

- Identifiers cannot be a C keyword
  You cannot use return as an identifier
  Complete list of C keywords given in your text

- Identifiers have no length limit

Caution:
  Several compilers support unlimited length identifiers
  Specify must be unique in the first 32 bytes
  This is not standard
  Refer to your compiler documentation

- 8 of 21 -
Simple Variables

Variable is
- Object we use to hold data
- Is an identifier
  That is it has a *name*
  So we can refer to it in our program
- Has a *type*
  Tells us how big the variable is
  Kind of like barrels
  We have a number of different types or sizes of barrels
  1, 2, 5, 10, 50 gallon containers
  Obviously each holds a different amount
  Also immediately obvious
  Trying to put contents of 50 gal barrel
  Into 1 gal barrel is problem

Intrinsic Types

Look around real world
- Have number of different units we measure by
  inches, feet, yards, miles
  Centimeters, meters, kilometers
  Gram, kilograms, etc.
- These are standards
- They are intrinsic units of measure

C language does same thing

We have number of standard numeric types
- These are predefined variable types
  Of different sizes

Such numeric types
- Called the *intrinsic types*,
  Sometimes called *fundamental types*
The classifies intrinsic types into two groups

- Integral Types
- Floating Point Types

Let's take a look at these

**Integral Types**

*Integral* or integer types are whole numbers

They have no fractional part

**Example**

25, -6

As we've noted amount of space occupied by *integral* type

Depends upon the *integral* type

The C language standard defines following *integral* types

Each type is of a different size

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>character</td>
</tr>
<tr>
<td>short</td>
<td>short integer – usually half size of integer</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>long</td>
<td>long integer – usually twice size of integer</td>
</tr>
<tr>
<td>bool</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Integral types may be either *signed* or *unsigned*

By default all the types identified above are signed

except *bool*

*Signed integer* can be either positive or negative

*Unsigned integer* can only be positive

**Signed Integer Types**

Defined with or without the C keyword *signed* in integer definition

**Example**

To define a signed integer with a value of -25 in our program

We write

```c
signed int aNumber = -25;
```

or simply

```c
-25;
```

- 10 of 21 -
int aNumber = -25;

We can use second form because
Language standard says
All integers are specified as signed by default

Unsigned Integer Types
Defined by using the C keyword unsigned in the integer definition

Example
To define an unsigned integer with a value of 25
We write
unsigned int aNotherNumber = 25;

Do not have a sign bit
Consequently, unsigned integers can only be positive

Why use an unsigned integer anyway?
Numbers stored in computer in binary
Each different type has different number of bits
For signed numbers
One of the bits used to indicate polarity of number
Typically the most significant bit – msb

➢ Writing the keyword signed or using default says
  Interpret the msb as sign
➢ Writing unsigned tells compiler
  Use all bits in word as data

char Integer Type
Historically used to represent characters

By assigning a value to a char
We are in effect assigning a character
char x = ‘A’;
char y = 65;
If we are using ASCII code

ASCII is the acronym for

American Standard Code for Information Interchange

Refer to asciitable.com for the ASCII Character Set

Both of above lines of code do the same thing

More generally

Denote a character by enclosing the character in single quotes

**Example**

print3.c

The code fragment below should display the letter M on the screen.

```c
#include <stdio.h>
int main(void)
{
    printf("M \n");
    return 0;
}
```

Some older compilers would

Convert the character M to its numeric value and

In the ASCII Character Set

Display the number on the screen instead of the letter

In this case 77

Try the simple program above on your computer

To verify that the letter M appears on the screen

From what we have seen so far

There is a difference between 77 and 'M'

77 is the char integer value and

'M' is the glyph corresponding to 77

In the ASCII character set
When I say 0 ....

- Similarly there is difference between 0 and '0'
- Failure to notice the difference
  - Can result in a program bug

0 is integer zero
  - That is, all bits are 0

A char integer with a value of zero
  - Often referred to as a null character

On the other hand the character '0' has a value of 48
  - In the ASCII Character Set

If we intend to compare something to zero and we use '0'
  - We are actually comparing to 48

Always be certain whether you mean
  - Value of zero or the character '0'

**bool** Integer Type

- bool integer type can contain only two values
  - zero (false) or one (true)

Explicit Boolean type added to ANSI standard in March 2000
  - Some compilers don't recognize it as intrinsic or native type

Should use the keywords *true* and *false*
- When setting a bool variable
  - Rather than using 0 or 1.
Example
To express conditions for success or failure of some test
We might write
\begin{verbatim}
    bool pass = true;
    bool fail = false;
\end{verbatim}

,bool variable is commonly referred to as a switch
Because it can only be
true or false
one or zero
ON or OFF

How Big is Big…Size of Integer Types
Each of integral types
Can be of a different size
That is requires different number of bits of memory
To store value

The smaller the size of the variable
Fewer bits allocated to store it
Smaller the value it can hold

Sizes of integer types varies by
Computer and operating system

One way to find the size of these types on your computer
Run the following simple program…

Example
variables0.c

\begin{verbatim}
#include <stdio.h>
int main(void)
{
    // checking sizes
    printf("A char is %d byte \n", sizeof(char));
    printf("A short is %d bytes \n", sizeof(short));
    printf("A int is %d bytes \n", sizeof(int));
\end{verbatim}
printf("A long is %d bytes \n", sizeof(long));
    return 0;
}

On my machine this displays
    A char is 1 byte
    A short is 2 bytes
    A int is 4 bytes
    A long is 4 bytes

Code sample uses sizeof operator
    Operator will always return Number of bytes of the type
        Between the parentheses

We can find same information
    By looking in header files
        limits.h
        float.h

Minimum and Maximum Integer Values
    To find the minimum and maximum integer value
        Integer type can hold

We can include the limits header file in our program

Example
    variables1.c

    #include <stdio.h>
    #include <limits.h>
    int main(void)
    {
        printf("%d \n", CHAR_MAX); //displays char maximum
        printf("%d \n", CHAR_MIN); //displays char minimum
        printf("%d \n", SHRT_MAX); //displays short maximum
        printf("%d \n", SHRT_MIN); //displays short minimum
        printf("%d \n", INT_MAX); //displays int maximum
        printf("%d \n", INT_MIN ); //displays int minimum
printf("%d \n",LONG_MAX); //displays long maximum
printf("%d \n",LONG_MIN); //displays long minimum

return 0;
}

On my machine this displays

127
-128
32767
-32768
2147483647
-2147483648
2147483647
-2147483648

The limits.h header file
Contains the minimum and maximum values
Of the integer types.

Refer to your textbook
For the names of the minimums and maximums
Of the unsigned integer types.

Floating-Point Types
We have seen how we use integer types
To express information in computer

With integer type
Size of number we can express
Determined by number of bits in integer
Early PCs had integer containing 16 bits
Largest integer number that could be expressed
65,536

In early computers such restriction
Didn’t bother users
They were all mathematicians
Used to working with such limitations
Simply scaled the numbers
We could certainly use similar scheme
Let’s go for easier way

Clearly need to represent large numbers in computer
Have to keep track of national debt after all

To do so we use different data type
Called floating point type

Floating point type permits us to express numbers in mixed format

**Example**
Consider the number 3500
We can easily rewrite this as
0.35 x 10000
Next we rewrite 10000 as
10*10*10*10 or using a shorthand notation
10\(^4\)
Thus original number becomes
0.35 x 10\(^4\)
When we express a number in such a format
We call the 0.35 the fractional part or mantissa and
The 10\(^4\) the exponent part

When we store the number in memory
We’re only concerned with
0.35 the fractional part
4 the actual exponent

When number expressed in such a format
We say we’re using a floating-point format
Other terms that may be used include
    Exponential format
    Scientific notation

We state that such numbers are real numbers.

C language standard specifies two floating-point types
    float
    double

Each of these types contains the same maximum exponent
    Double allows for more significant figures
        More digits to right of decimal point

    There is even a long double
        Allows even more significant figures

Using floating-point numbers
    We can represent very large or very small numbers

    There is a downside
        When the maximum allowed number of significant digits exceeded
            Significant figures are automatically rounded

Now you have an approximation of original real number
    If we have built a large financial application
        Using floats and doubles
            May find books don’t balance
                For your very large corporate clients

Floating-Point Constants
    Floating-point constants by default are of size double

If we write
    float pi = 3.14159;
We get truncation warning
3.14159 has been created as *double*
Which is larger than a *float*

Compiler warns pi may not have expected value
We can avoid warning
By directing the compiler to create 3.14159
As a float and not a double.

```
float pi = 3.14159f;
```

or
```
float pi = 3.14159F;
```

Similarly long double constant can be written as
```
long double euler = 3.1415918281828459045L
```

**The const Qualifier**

When variable is created
We can *initialize* it to a value
This word is important
A variable can only be *initialized* one time
We can *assign* different values to it
Almost any time we want

**Example**

Consider the two declarations
These two variables are initialized
```
int legalPaperLength = 17;
int standardPaperLength = 11;
```

These two variables are then assigned to
```
legalPaperLength = 14;
standardPaperLength = 10;
```

In the second part of the example
Values assigned to variables
Don’t reflect actual sizes of paper
We can do this because
The two variables are just that
They are variable
We are permitted to change the value of a variable

In this case however
We want to ensure value cannot be changed
The value needs to be fixed to one value
We want the variable to be constant
Hmmm a constant variable - cool

C language gives us ability to do just this
To make the contents of a variable constant
We use `const` qualifier when we define the variable
Thus when we declare the variables we write

```c
const int legalPaperLength = 17;
const int standardPaperLength = 11;
```

We read these from right to left as

`legalPaperLength` is an integer constant or `standardPaperLength` is an integer constant

**Example**

Consider the two declarations
These two variables are initialized
```c
const int legalPaperLength = 17;
const int standardPaperLength = 11;
```

If we now try
```c
legalPaperLength = 14;
standardPaperLength = 10;
```

Compiler gets quite upset
Will produce an error
Stating attempt to change a constant value
**const** qualifier can be used with any C variable
Signifies that contents of the variable cannot be changed
After initialization

Consequently constant variable must be initialized
A value when the variable is defined

**Summary and Review of Objectives**
At this time you should have a general understanding of

- High level structure of a C program
- The basic building blocks of a C program
- Concept of identifiers
- Some of the data types that are part of the C language
- The sizes of the basic data types
  - ✓ Use data the user can enter into our program
  - ✓ Return computed results