References

Table of Contents

Session 2: Variables and Data Types

Variables and Expressions

Source	http://en.wikiversity.org/wiki/Variables_and_Expressions
Date of Retrieval	19/11/2012

Variables

In C, a variable is a named area of storage in memory. Memory is some form of physical hardware on a computer that either temporarily or permanently stores data. To that extent, variables are declared, defined and used to manipulate a stored value.

To use a variable it must first be declared, preceding the alphanumeric representation with a type. The type used to represent a variable is a constraint placed on the representation of the data's format and length in memory. Primitive data types in C, for instance, are restricted to storage in memory that is no greater than 8, 16, 32, or 64 bits. One example of a data type in C is int, which has a size suggested by the host machine architecture, for example, it may be restricted to 32 bits. In addition to the length, types are constrained by format. In this case, the int is limited to integer data. The declaration of a variable, then, starts with the data type, and is followed by the identifier (the alphanumeric name). To indicate the end of the declaration, use a semicolon. Like this:

int a;

In this example, 'int a;', has a data type of 'int' and an identifier, 'a'. The variable 'a', can store a value no greater than the value specified in INT_MAX - a value defined in the header file <limits.h>. If the width of an int happens to be 32 bits, then INT_MAX will be 2147483647. The value currently assigned to the variable 'a', is undefined, but is constrained to an integer format.

In C, you can assign a value to a variable when defining it. To do so, use the assignment operator, '='. The assignment operator should not be confused with the mathematical sign of equality. In mathematics it is used to indicate equality; however, in C, it is used to assign a value to a variable. So, the definition of a variable, in C, begins with the variable identifier, followed by the assignment operator, and then the initializer (e.g. an integer when assigning to an integer type). Like the declaration, to indicate the end of the definition, use the semicolon. Here is an example:

a = 16;

In this example, the value 16 is assigned to the variable 'a'.

Expressions

To manipulate the variable, 'a', declared and defined in the previous section, an expression is needed. By definition, an expression, in C, is an interpreted combination of values, variables, operators or functions. There are a number of operators available including addition, '+', subtraction, '-', division '/', and multiplication '*'. In an expression, the variable name on the left side of the assignment operator represents the area of memory that stores interpreted results. Variable and constants on the right side of

the assignment operator are interpreted to determine a result prior to assignment. Note these definitions and declarations:

int a;

int b;

a = 0;

b = 8;

What follows is a statement which manipulates storage in memory (an expression becomes a statement when it is followed by a semicolon):

a = b + 24;

In this statement, the constant ‘24’, is added to the value stored in the variable ‘b’. The result of that calculation, then, is assigned to a memory location, symbolically represented by the variable ‘a’. After the interpretation of the statement, the variable 'a' is assigned the value 32.

~~~ End of Article ~~~

Standard Libraries

Source	http://en.wikibooks.org/wiki/C_Programming/Standard_libraries
Date of Retrieval	19/11/2012

The C standard library is a standardized collection of header files and library routines used to implement common operations, such as input/output and character string handling. Unlike other languages (such as COBOL, Fortran, and PL/I) C does not include builtin keywords for these tasks, so nearly all C programs rely on the standard library to function.

History

The C programming language previously did not provide any elementary functionalities, such as I/O operations. Over time, user communities of C shared ideas and implementations to provide that functionality. These ideas became common, and were eventually incorporated into the definition of the standardized C programming language. These are now called the C standard libraries.

Both Unix and C were created at AT&T's Bell Laboratories in the late 1960s and early 1970s. During the 1970s the C programming language became increasingly popular, with many universities and organizations beginning to create their own variations of the language for their own projects. By the start of the 1980s compatibility problems between the various C implementations became apparent. In 1983 the American National Standards Institute (ANSI) formed a committee to establish a standard specification of C known as "ANSI C". This work culminated in the creation of the so-called C89 standard in 1989. Part of the resulting standard was a set of software libraries called the ANSI C standard library.

Later revisions of the C standard have added several new required header files to the library. Support for these new extensions varies between implementations.

The headers <iso646.h>, <wchar.h>, and <wctype.h> were added with Normative Addendum 1 (hereafter abbreviated as NA1), an addition to the C Standard ratified in 1995.

The headers <complex.h>, <fenv.h>, <inttypes.h>, <stdbool.h>, <stdint.h>, and <tgmath.h> were added with C99, a revision to the C Standard published in 1999.

Note:

The C++ programming language includes the functionality of the ANSI C 89 standard library, but has made several modifications, such as placing all identifiers into the std namespace and changing the names of the header files from <xxx.h> to <cxxx> (however, the C-style names are still available, although deprecated).

Design

The declaration of each function is kept in a header file, while the actual implementation of functions are separated into a library file. The naming and scope of headers have become common but the organization of libraries still remains diverse. The standard library is usually shipped along with a

compiler. Since C compilers often provide extra functionalities that are not specified in ANSI C, a standard library with a particular compiler is mostly incompatible with standard libraries of other compilers.

Much of the C standard library has been shown to have been well-designed. A few parts, with the benefit of hindsight, are regarded as mistakes. The string input functions gets() (and the use of scanf() to read string input) are the source of many buffer overflows, and most programming guides recommend avoiding this usage. Another oddity is strtok(), a function that is designed as a primitive lexical analyser but is highly "fragile" and difficult to use.

ANSI Standard

The ANSI C standard library consists of 24 C header files which can be included into a programmer's project with a single directive. Each header file contains one or more function declarations, data type definitions and macros. The contents of these header files follows.

In comparison to some other languages (for example Java) the standard library is minuscule. The library provides a basic set of mathematical functions, string manipulation, type conversions, and file and console-based I/O. It does not include a standard set of "container types" like the C++ Standard Template Library, let alone the complete graphical user interface (GUI) toolkits, networking tools, and profusion of other functionality that Java provides as standard. The main advantage of the small standard library is that providing a working ANSI C environment is much easier than it is with other languages, and consequently porting C to a new platform is relatively easy.

Many other libraries have been developed to supply equivalent functionality to that provided by other languages in their standard library. For instance, the GNOME desktop environment project has developed the GTK+ graphics toolkit and GLib, a library of container data structures, and there are many other well-known examples. The variety of libraries available has meant that some superior toolkits have proven themselves through history. The considerable downside is that they often do not work particularly well together, programmers are often familiar with different sets of libraries, and a different set of them may be available on any particular platform.

ANSI C library header files

<assert.h> Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.

<complex.h> A set of functions for manipulating complex numbers. (New with C99)

<ctype.h> This header file contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).

<errno.h> For testing error codes reported by library functions.

<fenv.h> For controlling floating-point environment. (New with C99)

<float.h> Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).

<inttypes.h> For precise conversion between integer types. (New with C99)

<iso646.h> For programming in ISO 646 variant character sets. (New with NA1)

<limits.h> Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).

<locale.h> For setlocale() and related constants. This is used to choose an appropriate locale.

<math.h> For computing common mathematical functions

-- see Further math or C++ Programming/Code/Standard C Library/Math for details.

<setjmp.h> setjmp and longjmp, which are used for non-local exits

<signal.h> For controlling various exceptional conditions

<stdarg.h> For accessing a varying number of arguments passed to functions.

<stdbool.h> For a boolean data type. (New with C99)

<stdint.h> For defining various integer types. (New with C99)

<stddef.h> For defining several useful types and macros.

<stdio.h> Provides the core input and output capabilities of the C language. This file includes the venerable printf function.

<stdlib.h> For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.

<string.h> For manipulating several kinds of strings.

<tgmath.h> For type-generic mathematical functions. (New with C99)

<time.h> For converting between various time and date formats.

<wchar.h> For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages. (New with NA1)

<wctype.h> For classifying wide characters. (New with NA1)

Common support libraries

While not standardized, C programs may depend on a runtime library of routines which contain code the compiler uses at runtime. The code that initializes the process for the operating system, for example, before calling main(), is implemented in the C Run-Time Library for a given vendor's compiler. The Run-Time Library code might help with other language feature implementations, like handling uncaught exceptions or implementing floating point code.

The C standard library only documents that the specific routines mentioned in this article are available, and how they behave. Because the compiler implementation might depend on these additional implementation-level functions to be available, it is likely the vendor-specific routines are packaged with the C Standard Library in the same module, because they're both likely to be needed by any program built with their toolset.

Though often confused with the C Standard Library because of this packaging, the C Runtime Library is not a standardized part of the language and is vendor-specific.

Compiler built-in functions

Some compilers (for example, GCC) provide built-in versions of many of the functions in the C standard library; that is, the implementations of the functions are written into the compiled object file, and the program calls the built-in versions instead of the functions in the C library shared object file. This reduces function call overhead, especially if function calls are replaced with inline variants, and allows other forms of optimization (as the compiler knows the control-flow characteristics of the built-in variants), but may cause confusion when debugging (for example, the built-in versions cannot be replaced with instrumented variants).

POSIX standard library

POSIX, (along with the Single Unix Specification), specifies a number of routines that should be available over and above those in the C standard library proper; these are often implemented alongside the C standard library functionality, with varying degrees of closeness. For example, glibc implements functions such as fork within libc.so, but before NPTL was merged into glibc it constituted a separate library with its own linker flag. Often, this POSIX-specified functionality will be regarded as part of the library; the C library proper may be identified as the ANSI or ISO C library.

~~~ End of Article ~~~

Session 15: Functions

Functions and Methods

Source	http://en.wikiversity.org/wiki/Functions_and_Methods
Date of Retrieval:	19/11/2012

Functions

In today's world we believe that Time is Money. The less time you spend in writing a program, the more efficient use of resources. Functions help us to avoid duplication of code and hence save time, cost and effort.

Let's say we would like to write a code that calculates the square of a number. The application that we are asked to design requires that you calculate the square of several numbers again and again. So, you will write the following code to calculate the square of the number wherever needed.

```
int result;
result = a * a;
```

If you have to calculate the square of the number at 3 different places in your program, you will have to write these lines of code over and over again.

```
void main()
{
___________;
___________;
result = a * a;
___________;
___________;
result = b * b;
___________;
___________;
result = c * c;
}
```

Instead of doing this over and over again, we can define a function -

```
int square(int n)
{
    int ans = n * n;
    return ans;
}
```

Once a function has been defined, we can now use this function wherever we would like to use it. For example,

```
void main()
{
___________;
```

```
   ___________;
   result = square(a);
   ___________;
   ___________;
   result = square(b);
   ___________;
   ___________;
   result = square(c);
   }
```

Thus, instead of writing the entire code over and over again, we can simply call the function here. Functions become the backbone of any C algorithm. Any C data type can be passed to a function (except an array or function - if passed they will be converted to pointers), and any C data type can be returned (except an array or function). Functions can also call other functions:

```
 int multiply_together( int n1, int n2 )
 {
     return n1 * n2;
 }
 int add_two_and_multiply( int n1, int n2 )
 {
     n1 = add_two( n1 );
     return multiply_together( n1, add_two( n2 ) );  /* returns (n1+2)*(n2+2)
*/
 }
```

You can call a function inside a function, as illustrated above.

Every C program has at least one function, that function being main(), which is the entry point of every program.

~~~ End of Article ~~~