A Librarian's Introduction to Programming: **C**

Tim Ribaric

One of the most enduring computer programming languages you're ever likely to encounter is C. Unlike other languages that were created in the 1970's, many applications are still actively developed in C. Contrast this longevity for example with code written in FORTRAN, many institutions with legacy systems have additional development costs allocated exclusively to make sure these systems continue to operate, mostly because new programmers do not spend time developing FORTRAN skills. In conrast code written in C doesn't have the same upkeep costs since programmers with an understanding of modern languages de facto have some workable knowledge of C.

In fact since C is a common ancestor for a series of newer languages including: C++, Java, Objective-C and Python, just to name a few. Developing an understanding of the syntax and semantics of C provides a great basis for understanding these new languages. However there is an interesting dichotomy floating around the industry. Some professional advocate that C should be the first language any programmer embarks up since it provides the right combination of syntactic structures and imparts an understanding of how memory is allocated and used. On the other hand some argue that writing good code in C is very difficult due to exactly the same rationale. In fact poorly written C might end up being a liability as it sometimes exposes very obscure security problem. The recommended resources has a brief sampling of this discussion.

As an information professional you'll probably never have a direct use-case for choosing the C language to develop an application or web service but having an understanding of it will inform everything else you'll do with programming as every language out there can classify itself as C-like or not C-like. That might seem like a tautology but the resonating characteristics of C do lend themselves to such comparisons. Think almost of C as Latin, modern programming and paradigms are built on the root words of C. Unless Latin however, C is not a dead language. In fact it is thriving now as much as it ever has in its 4 decade history.

History/Development

C was the result of the work of Ken Thompson and Dennis Ritchie who were employed at AT&T Bell Labs. During the years of 1969 – 1973 Thompson and Ritchie were working on a language to re-implement the UNIX operating system, this work first began using a language called B. The result was a great success as it allowed UNIX to be written in hardware agnostic language C. For example a development team just needed to write a C-compiler in Assembler and then compile the C code using that compiler. Up until this point an operating system was written in the lowest level language called Assembler. This means that the operating system is tied very heavily to the Assembler used to write it. So for example if a new chip came out with more memory on it the Assembler would have to be rewritten to accommodate the new chip, meaning essentially a rewrite of the operating system. In this new development paradigm after a new chip is invented you simply need to write a new C compiler in your Assembler to build UNIX for that hardware. This is considerably easier then rewriting the whole operating system.

As years progressed however C began to undertake different authoritative versions as computer hardware and understanding evolved. The underlying mechanisms however were consistent through these iterations, so knowing one type is sufficient to know about them all. The major epochs were:

K & R – Named after authors of the original language specification Kernighan and Ritchie, first see in the early 1970's.

ANSI C – Came into existence officially in 1989 and was the first attempt to create a more complete standardized version of C so that it may start to be used with the new fledging microcomputer market. Up until this point C was mostly for mainframe use and the desktop computer was slowly finding its way into homes.

C99 – First finalized in 1999 its goal was to introduce a new collection of features that allowed for things like in-line functions and variable-length arrays. Features that were shipping with other lanaguages there were being developed at this time.

C11 - Work on this version first began in 2007 but the first official specification was published in 2011. Much like the C99 iteration this version of C sought to add more features to the language.

It is fascinating to note over the course of 40 plus years C is still being actively expanded and improved. This is testament to the ubiquity and usefulness of the language.

What then about the descendants of C? Knowing that C is a general purpose language useful for implementing UNIX what/when did its progeny first begin? A comprehensive look would take volumes but some of the highlights:

C++ - An extension of C that introduced a programming paradigm called Object Oriented, the specification of this was first released in 1983.

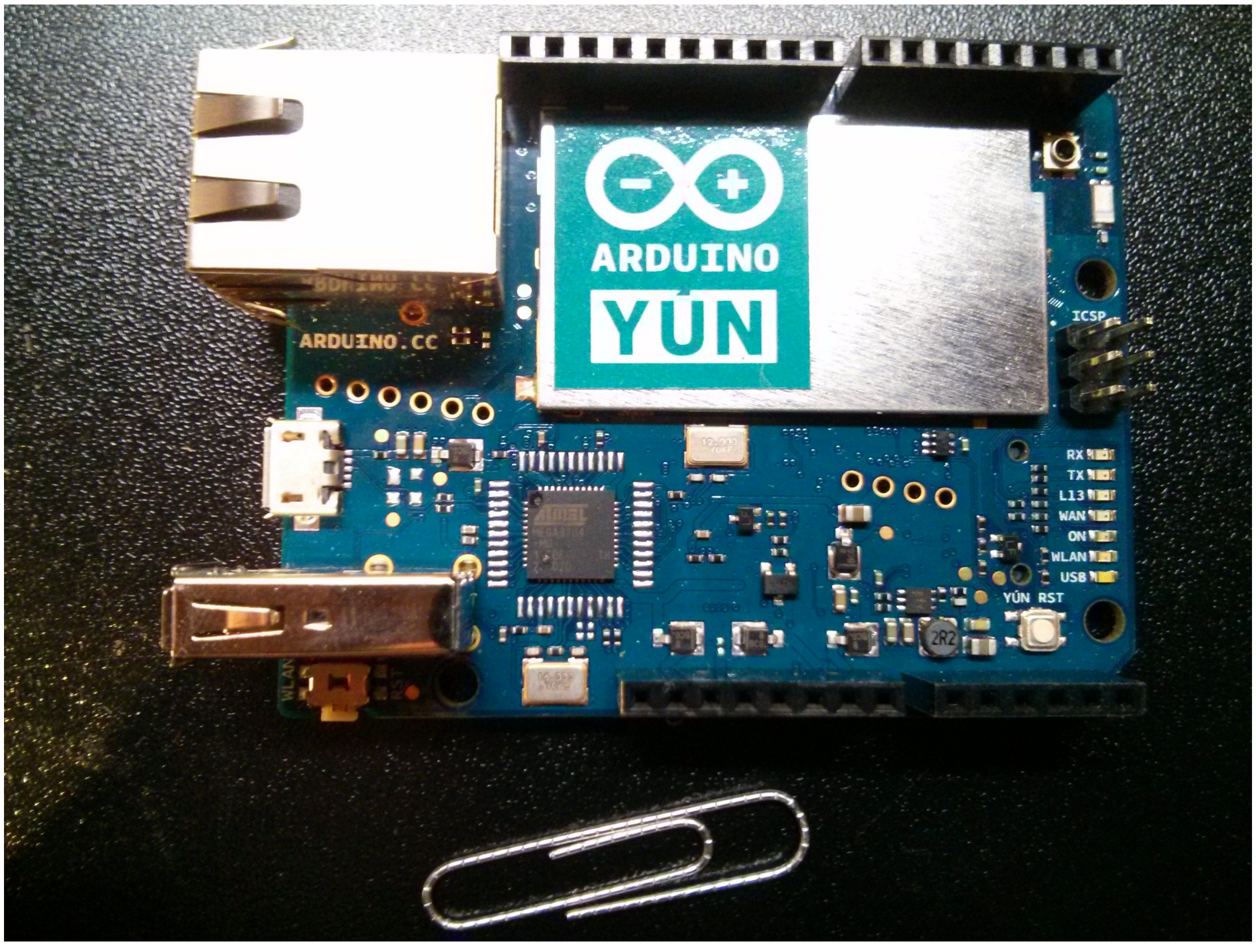
C# - Pronounced as if were a music note: C-sharp. It can be seen as a progression of both C and C++ developed by Microsoft to be utilized on their proprietary .NET platform. It was first released in 2000.

Objective-C – A slight extension of the C language heavily used by Apple for the OS X and iOS operating systems, also first seen in 1983.

Python – Quite often Python is the language of choice for those interested in embarking on a learning to coding project, especially in the past few years. It is a language built on top of C whose primary aim is to create code that is human readable that can encapsulated very complex operations in short snippets of code. Python was first introduced in 1991. Having a good understanding of C allows a coder to really appreciate the beauty and complexity that Python programs are capable of.

Uses

As mentioned previously the initial use of C was to implement the UNIX operating system. However as time progressed C demonstrated that it is a very capable application great for writing all sorts of applications. C requires a large amount of understanding from the programmer who chooses to use it. The language doesn't provide a completely structured environment where all modules/libraries are interactively included when needed. However not all C use is esoteric or operating system specific. One interesting field where knowledge of C is handy is with the Arduino microcontroller. The Arduino is a miniature credit card sized board that has been one of the key components of the recent maker surgence.



A picture of the modern Arduino board called Yun. It's defining feature is that is that has build in Wifi. The paper clip is to demonstrate scale.

An Arduino is a device that let you make hardware devices that interface with your computer through a variety of different components attached to the board. Think, switches, buttons, LCD displays and infrared emitters. In the image above these sensors are plugged into the series of black plugs. The code you write for the Arduino is more or less a flavor of C.

Pros and Cons

A programming language can't really enjoy a 40 year lineage without have some endearing qualities. On the downside these good qualities are pretty specific.

Pros

Fast and Utilizes hardware well - The primary and obvious benefit to using C is that is utilizes the computer/machine it is working on very well. Due to this fact you can find C programs running on embedded systems that are running UNIX or some variation of it such as Linux. The Internet router in your home is most likely running a version of Linux on a low powered processor. Sending our radio waves through the wifi antennas is probably made possible by some sort of C code. OpenWrt a popular after market operating system for home routers is more or less a version of Unix that ships with utilities written in C. OpenWrt is the operating system powering the LibraryBox, which is a standalone wifi powered file sharing outpost.

Extensive Support and existing products – Because C has been around for so long chances are there is a library already written that your program can us. For example if you are writing some software that uses cryptography in some way you can utilize a library called cryptlib which is written in C and can perform all the complicated math you'll need. In fact many Open-Source applications are compiled C products. For example OpenSSL a software packed used to ensure the https traffic in your web browser is a C program.

C is ubiquitous - While you code and create programs in other languages the underlying compiler that that language uses might in fact be written in C. As mentioned earlier the Python programming language is itself an off-shoot of C. With a little bit of work it is possible to compile a Python program to a corresponding C program, which quite often is done so the code runs quicker. Consider also the extremely popular and ubiquitous web language PHP. The underlying software used to interpret the language is a platform called Zend Engine and which is written in C.

Cons

Integration into a Web Environment – Today most budding coders are interested in creating web based services. That is to say the most popular form of development these days seems to be making apps that are accessed via the browser. This type of development with C is pretty much a non-starter, however it is not a complete impossibility. If pressed you could utilize a 'Common Gateway Interface (CGI)' program where a web server could render a web page and include the output of a C program in the final HTML. While C is a possibility with CGI programming PERL is often the language of choice for this kind of setup. Couple this further with the idea that CGI web programming in general is never really a first choice for web developers. There are other more reliable ways to generate dynamic web content, unfortunately C isn't well suited for this.

GUI Based Applications – Since C was born on the UNIX command line it is mostly capable of creating programs for the command line. If you'd like to create a desktop application to run in a GUI you'll need to spend some time learning how to use an external library that renders windows in the interface. This could be something along the lines of GTK+. Contrast this for example to Visual Basic which allows you to interactively draw widgets and forms and add code in a fluid interactive process.

Environment and Setup

Putting an environment together to write and compile C programs is a bit different for each operating system. In a Linux / Mac OS environment the basic tools come right out of the box, conversely in a Windows environment some additional tools need to be installed on the computer. This difference originates in the lineage of these two categories of Operating Systems. UNIX and its decedents are something called POSIX compliant. The acronym stands for Portable Operating System Interface, which is a standard written by the IEEEE that outlines some must haves for an operating system. Most of these prerequisites have to do with C implementation details and other specific Linux utilities. The way Windows evolved was separate and distinct from UNIX, so POSIX compliance was never a driving factor with that operating system. The important thing to remember is that the main goal of the POSIX standard is to ensure compatibility between systems. This goes hand in hand with the previously stated original objective of C to be a language that can be used to build UNIX through a compilation of C code.   The short of it is that POSIX compliant operating systems implicitly ship with all the tools you need to write C code. Windows on the other hand is not a POSIX compliant operating system and getting the basic tools together requires the installation of a piece of software that makes your Windows computer behave more like a UNIX based machine. Now you wouldn't want to try to start writing enterprise level code after setting up using the methods described below, this is strictly a crash course that will provide you with just a taste of what C is and how it differs from other languages.

Linux / Mac OS

As mentioned previously the tools are available out of the box. Our investigation will use two tool:

`gcc` – which stands for gnu c compiler. If you open a terminal window and type 'gcc' and hit enter you'll probably see some error and message that 'compilation terminated'. Great that was easy.

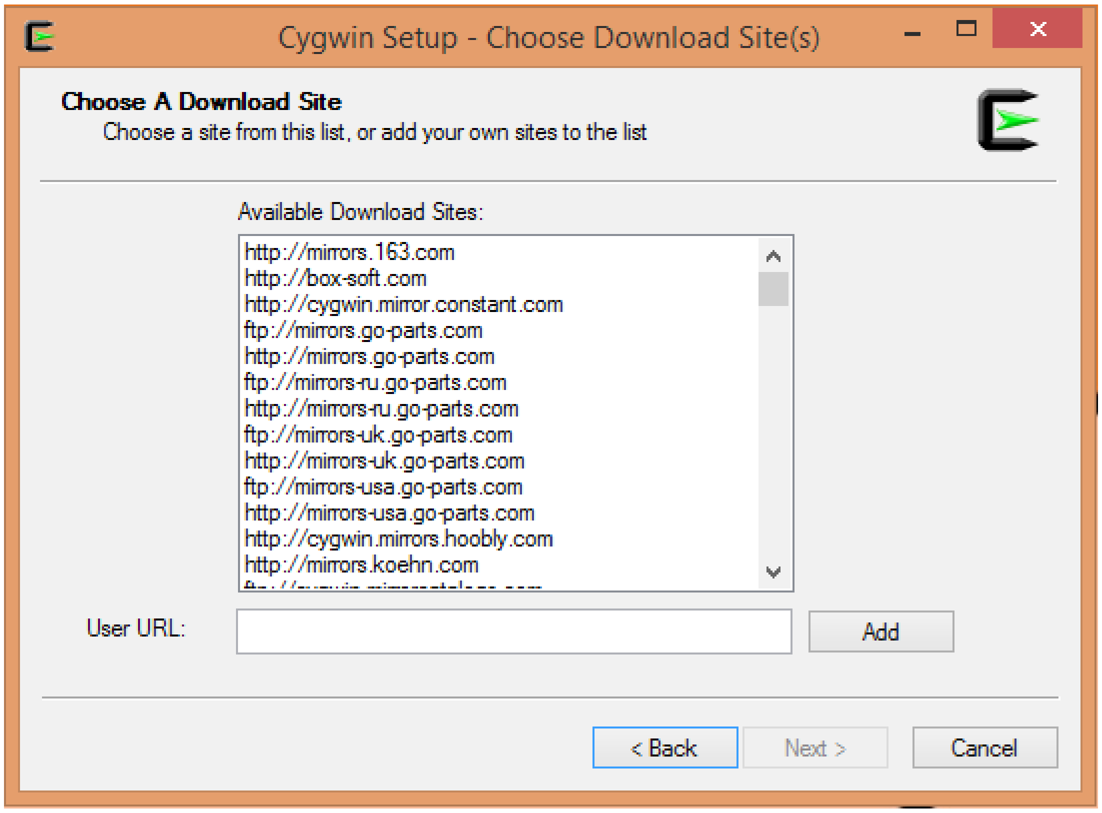
`nano` – Any text editor will do to enter code. We'll use 'nano'. In both Linux (in this case Ubuntu) and Mac OS it comes pre-installed. To verify open a Terminal and type in `nano`. You'll see a rudimentary interface drawn to the screen. Hit Ctrl + X to exit.

Done. Compare this now with the instructions for Windows.

Windows

A version of the C tools you'll need doesn't really ship with Windows out of the box. If for example you have installed some version of Microsoft's Visual Studio it will come with the tools we'll need. In our case we are going to do something different. We are going to install a utility called CygWin. CygWin is essentially a piece of software that will help you to recreate a Linux environment in Windows. The added benefit of CygWin is that it will allows you to run other interesting and very helpful Unix shell commands like `more` and `echo`. More on these later. Traditionally speaking learning C goes hand in hand with UNIX, you would be best served to spend some time getting comfortable on a command line to get a full flavor of the experience.

To begin download the correct version of CygWin from http://cygwin.com/install.html. To make things easier try the 64 bit release first and then try the 32 bit if that doesn't work. Once you run the file you downloaded you'll be prompted with a lot of click throughs. You'll need to pick a Download Site from this screen, any one will do:



Choose A Download Site

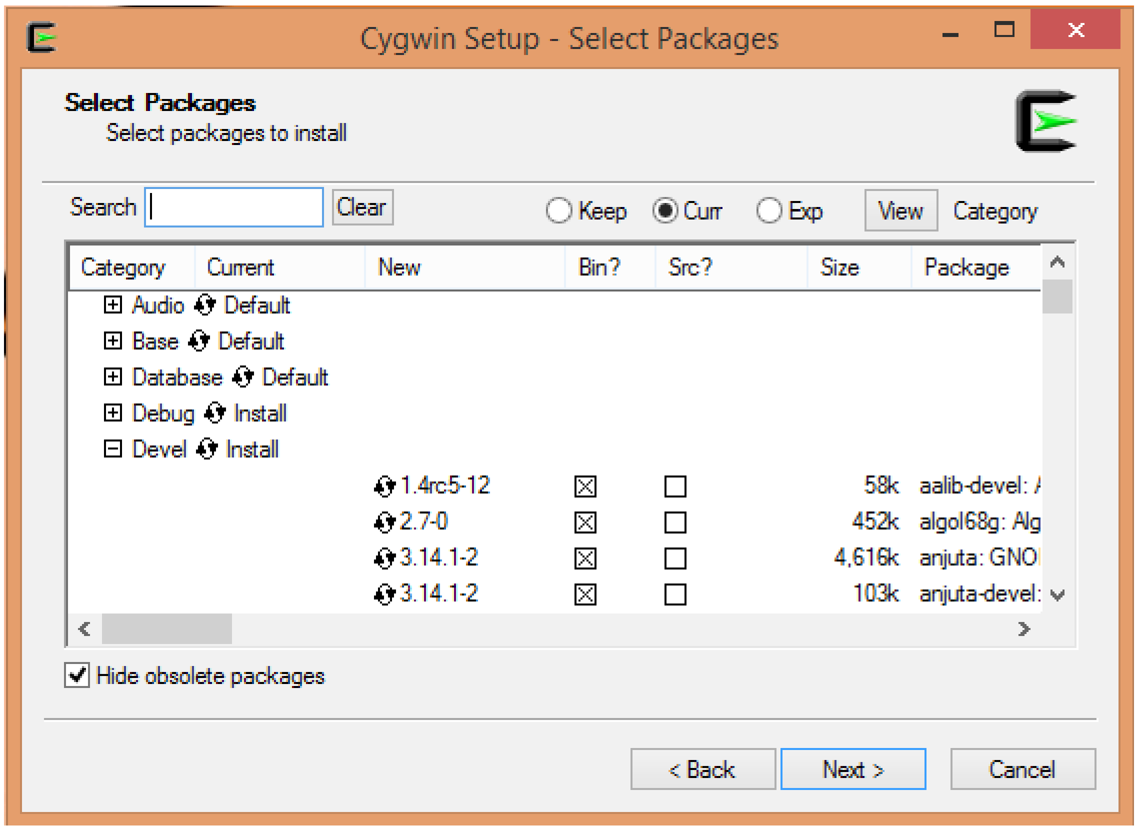
Choose a site from this list, or add your own sites to the list

Available Download Sites:

```
http://mirrors.163.com
http://box-soft.com
http://cygwin.mirror.constant.com
ftp://mirrors.go-parts.com
http://mirrors.go-parts.com
ftp://mirrors-ru.go-parts.com
http://mirrors-ru.go-parts.com
ftp://mirrors-uk.go-parts.com
http://mirrors-uk.go-parts.com
ftp://mirrors-usa.go-parts.com
http://mirrors-usa.go-parts.com
http://cygwin.mirrors.hoobly.com
http://mirrors.koehn.com
```

User URL:

You can also pick a site that is geographically close to help speed up the download. Just look for domain names that match your locale. Once the base system is installed you'll be prompted to install what packages you want. There are many options available here but all we want to do is expand the 'Devel' box and click on the double arrow so that everything will be installed. It should look like the following:



Cygwin Setup - Select Packages

Select Packages
Select packages to install

Search [ ] Clear   ○ Keep ● Curr ○ Exp View Category

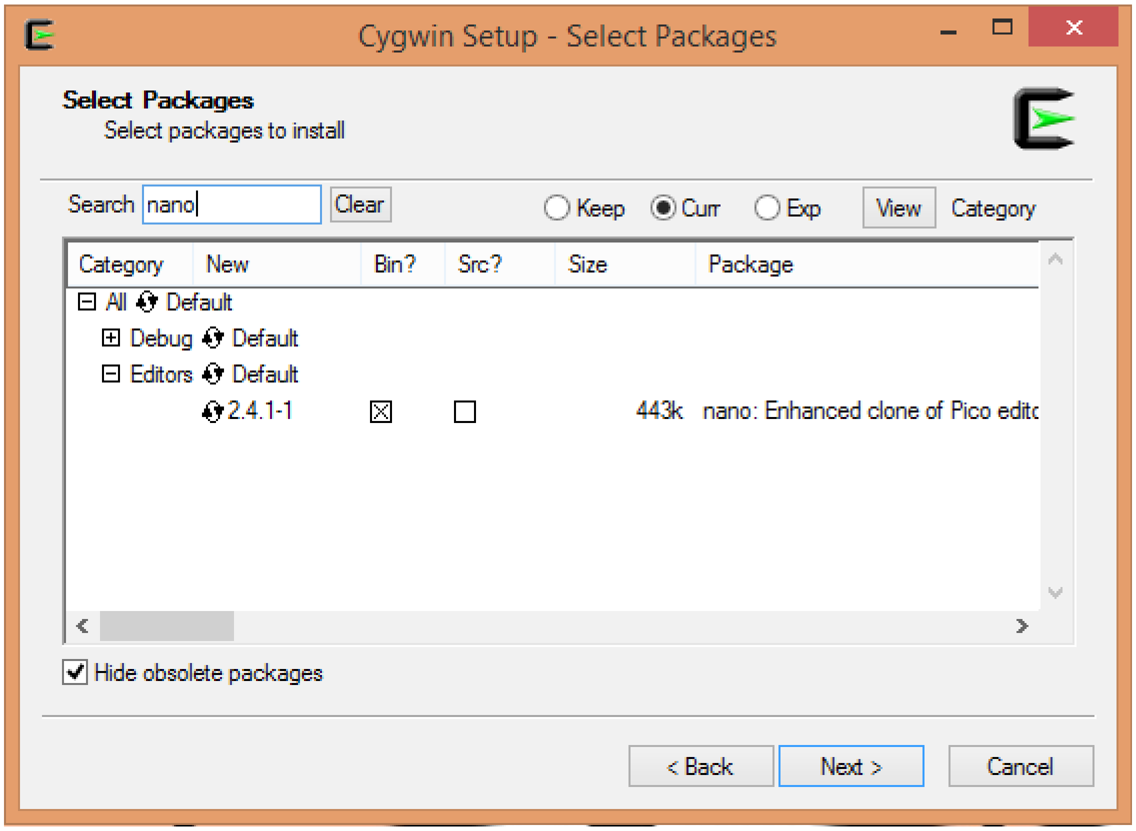
Category	Current	New	Bin?	Src?	Size	Package
⊞ Audio Default						
⊞ Base Default						
⊞ Database Default						
⊞ Debug Install						
⊟ Devel Install						
		1.4rc5-12	☒	☐	58k	aalib-devel: A
		2.7-0	☒	☐	452k	algol68g: Alg
		3.14.1-2	☒	☐	4,616k	anjuta: GNO
		3.14.1-2	☒	☐	103k	anjuta-devel:

☑ Hide obsolete packages

< Back   Next >   Cancel

The box under 'Bin?' changes from an empty square to a checked square.

Now search for ‘nano’ expand ‘Editors’ click on the ‘Skip’ to toggle it so that it is a checked square. It will look like the following:



Cygwin Setup - Select Packages

Select Packages
Select packages to install

Search [nano] Clear ○ Keep [● Curr] ○ Exp View Category

Category	New	Bin?	Src?	Size	Package
⊟ All Default					
⊞ Debug Default					
⊟ Editors Default					
	2.4.1-1	☒	☐	443k	nano: Enhanced clone of Pico editc

☑ Hide obsolete packages

< Back Next > Cancel

Now hit ‘Next’ a couple of times, wait for the download and the install, hit ‘Finish’ at the last step to complete the process. Look for 'Cygwin Terminal' or similar in your start menu. The result is a box with a blinking cursor ready for you to type away.

Examples

For the sake of simplicity the example following will be shown using screen shots from a computer running Linux. Before we can begin with some actual examples, first a general overview of the steps in actually creating a C program:

- Write the source code of the application. Source code has the extension of c
- Compile the source code into an intermediate ‘object file’ that is something halfway between the human readable code and the binary instructions that the computer will run
- Link the newly generated ‘object file’ with additional ‘object files’ that introduce the necessary built in library functions required by the application. For example, if your program includes code that manipulates files you’ll need to link to object files that provide this functionality. The human readable version of these built in functions are kept in header files that have an extension of h
- The final step is to create an executable file that can be run standalone on the terminal command line

In our examination we'll be abbreviating some of these steps to make things a bit easier to follow.

Examples 1 - Hello World

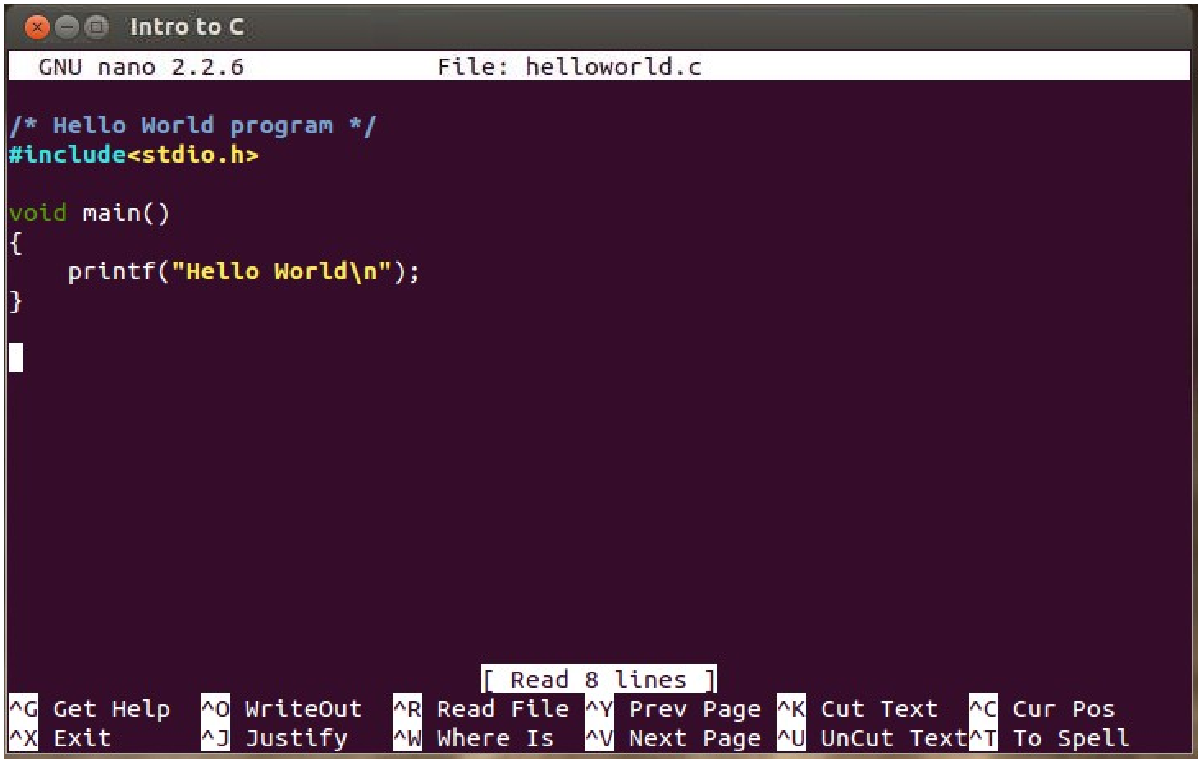
Open a Terminal window (or Cygwin depending on your operating system) and type the following:

```
nano helloworld.c
```

You'll be greeted with that rudimentary text editor interface. Type in the following exactly as it is written:

```
/* Hello World program */

#include<stdio.h>
main()
{
    printf("Hello World\n");
}
```

It should look like the following:



```
  GNU nano 2.2.6              File: helloworld.c

/* Hello World program */
#include<stdio.h>

void main()
{
    printf("Hello World\n");
}

                                [ Read 8 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Hit Control + O to save the file. Then Control + X to exit.

Once you have returned to the command line type the following:

```
gcc helloworld.c
```

This will compile and link your program into an executable file.

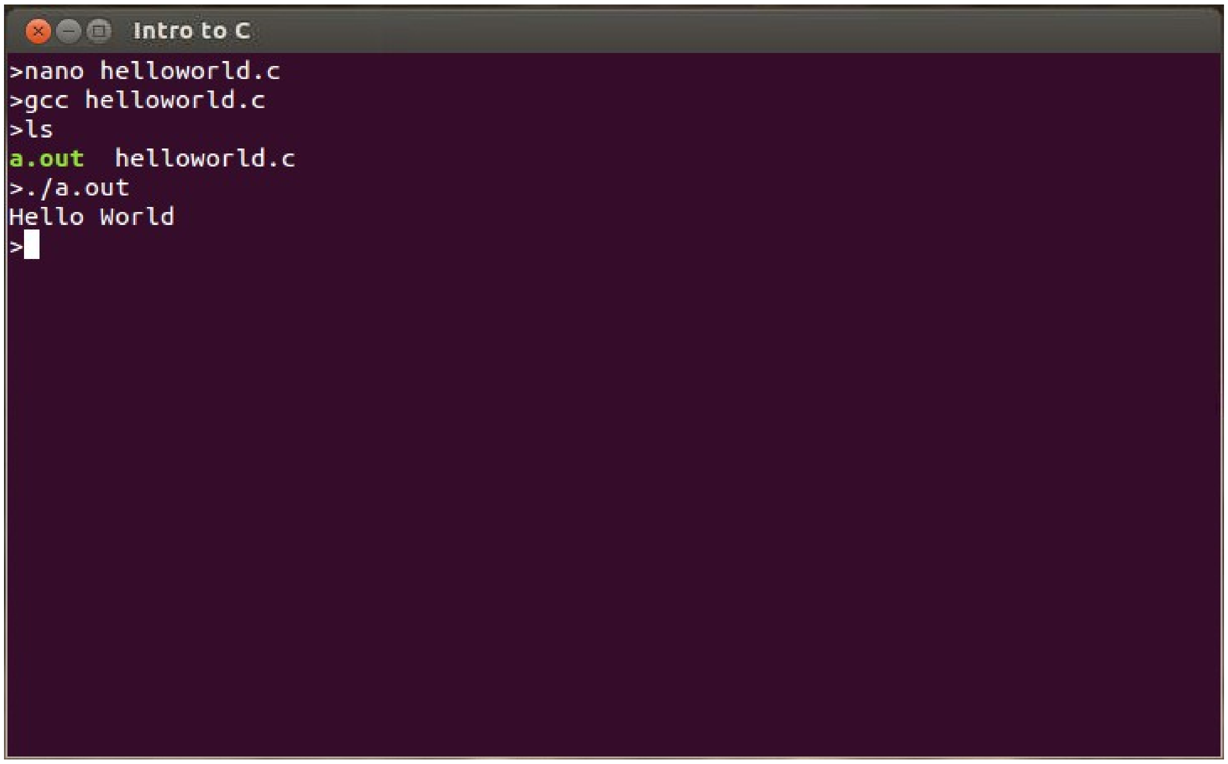
```
ls
```

This will show you the contents of the current folder/directory.

```
./a.out
```

This will run your program. `a.out` is the default program name that gcc compiles to. We prepend `./` to the beginning of the command because we are saying look in the present directory first. The . is a shortcut operator the references the current directory. When we want to refer to something that is in a directory we use the / character.

After all of this wizardry only the most basic of tasks has been completed. Here is the output of the command terminal following all of these steps:



```
>nano helloworld.c
>gcc helloworld.c
>ls
a.out  helloworld.c
>./a.out
Hello World
>
```

Before looking at the actual code line by line it is worth pointing out the tight coupling between Linux and C. Now to the code listing. Let's go line by line:

Line 1. – This is a comment line. Anything written between /* and */ is there for the sake of the human reading the code. The C compiler just skips that line.

Line 3. - This is an include which essentially tells C to include the libraries for standard input and output. Every program you write that does anything interesting will require this line. It is a header file that lets you use functions like pritntf that we will see later. Put simply a header file is a human readable version of the 'object code' of those input/output functions.

Line 4. - Starts the main function of the program. It is proceeded with a the keyword void. Void is reserved word in C and indicates that the function won't provide any returning value. We'll see an example later where we want that main function return an integer. Anytime you write and compile a C program it will look for the main function and run whatever it finds there first.

Line 5 - Is the opening brace of the main function. Anytime you define a function in C you need an opening and closing brace character. The great things is that C doesn't care about indentation or white space. That only exists for the sake of the humans reading the code.

Line 6 – The payload of this program. This printf statement is a function that takes the supplied argument (in our case the quintessential introductory statement every programmer begrudgingly learns) and prints it to standard output. In most cases the screen. The bit about \n at the end of the string is an escaped character that appends a newline to the end of the string. After printf is you'll see a semi-colon. Every line of code in C needs to be terminated with a semi-colon, with the exception of the include line.

With this obligatory example of out of way let's delve into something particularly C like.

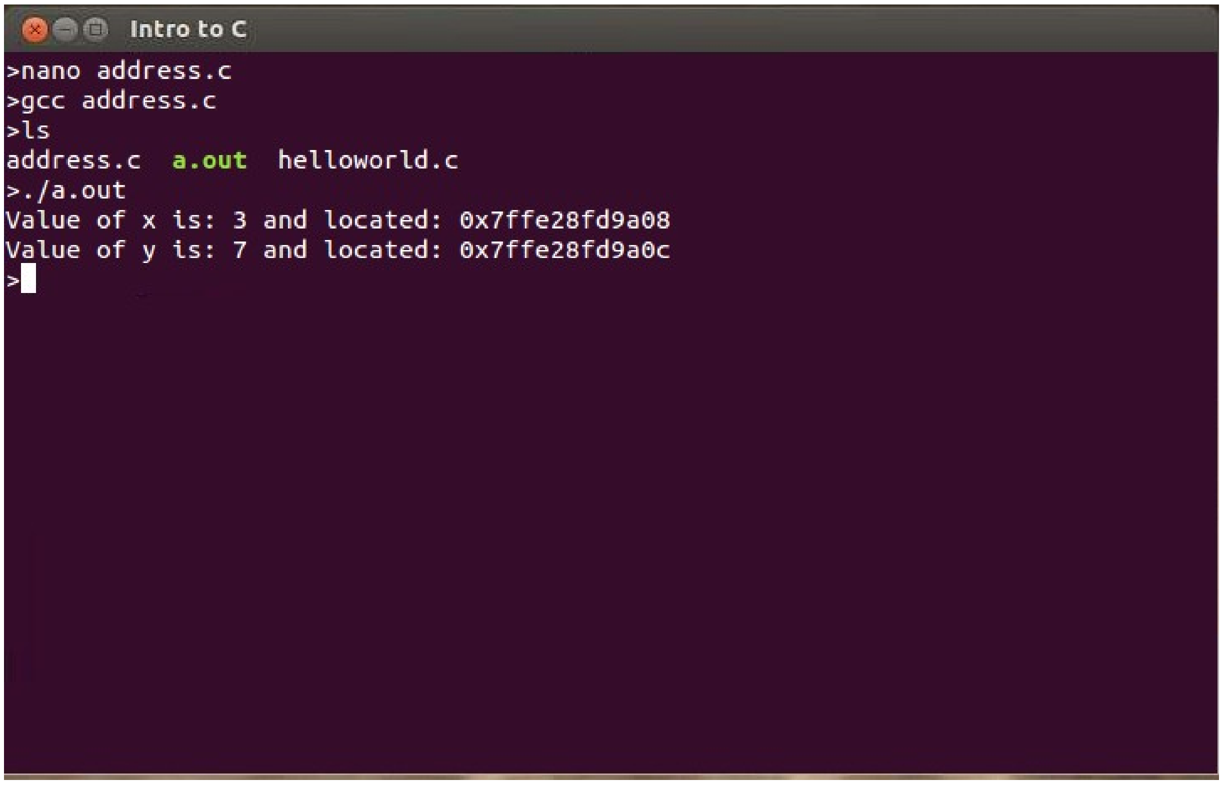
Example 2 – Addresses of values

Back at the command prompt trying using nano to start a file called address.c and then enter in this text and save the document:

```
/* Looking at the address of a variable */

#include <stdio.h>
```

```
main() {

int x;

int y;

x = 3;

y = 7;

printf("Value of x is: %d and located: %p\n",x,&x);

printf("Value of y is: %d and located: %p\n",y,&y);

}
```

The command line work flow should for this example should look like the following:



```
>nano address.c
>gcc address.c
>ls
address.c  a.out  helloworld.c
>./a.out
Value of x is: 3 and located: 0x7ffe28fd9a08
Value of y is: 7 and located: 0x7ffe28fd9a0c
>
```

Your output will be different then what is in the image. A line by line account of the code now, let's skip the parts we saw with the first bit of code.

Line 6. & 7. - Here we are declaring two integers. A fundamental data type of any language.

Line 8. & 9. - We assign some values to those newly created integers

Line 10. & 11. - Here is the C in action. We once again use printf to print some values to the screen. Printf is a pretty versatile function that does more then you can imagine. If you provide it a string of text with a % character embedded into it it will take other variables provided as arguments into the spot where the % is located. In our first example we print x using the %d sequence. Another C-ism in action, C doesn't really know what is in any memory address so by doing %d you are telling the interpreter to print the contents of that memory address as if it were an integer. (You'd use %s if you had a string you'd want to put into the sequence, %f for float etc) The second part is a %p which stands for pointer. The third argument of printf is &x which is actually the address in memory where x resides. The & operator tells C you want the address of the thing you are looking at, not the value of the thing.

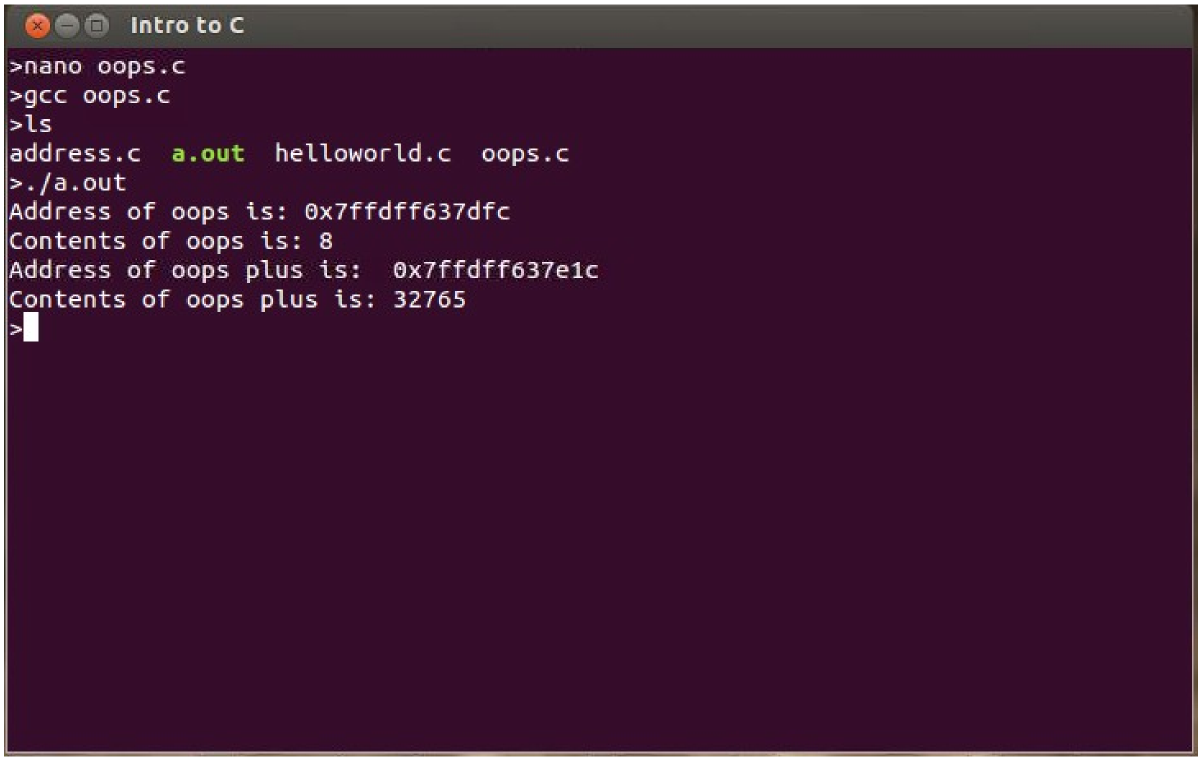
The purpose of this example is to show you just how low in level you can get with C code. When we print the two integers we first print their value, then we do something really obscure, we print the address of the bit of memory where that value is storied (for x this is `0x7ffe28fd9a08`) This ability to look at the address of where something is located is a fundamental piece of C, this mechanism is called a pointer. Understanding pointers is a lifelong pursuit and anyone who claims to have a great understanding of them really doesn't. Since C doesn't have object oriented features (or highly abstracted data structures like other languages such as Python) when you want to do something complex you usually pass a pointer to the start of your complex thing. For example an array of n integers is actually just n sequential spots in memory that are the size of integers long. When you want to reference the third item in the array you basically reference the contents of the memory located at an offset of plus 2 from the memory address of the first item in the array. Recall what was said earlier about bad C creating problems. If you are accessing the contents of memory directly, it might very well be possible that you write or read a value you weren't supposed to. Before moving to the next example it is worth looking at the difference between the two memory addresses of the integer variables. These numbers are written in a notation called hexadecimal. So you count from 0 – 9 and then A through F before adding a place holder. For example when you are counting in decimal (our usual method) when we go from 9 to 10 we start a new column (the 1) followed by the 0. In hex you start that new column after F. Just to keep thing from getting even more confusing we use a prefix of 0x every time we write a hexadecimal number. In this code we have two integers which most likely have been placed in consecutive spots in the computer's memory. The difference between these two addresses is 0x4. Therefore an integers takes 4 bytes of memory. Yup C is all about this kind of stuff. Great, why should we care? The next example tries to illuminate this.

Example 3 Oops.

Use nano again to edit the file oops.c

```
/* Accessing a bit of memory */
#include <stdio.h>


main () {


int oops;
oops = 8;


printf("Address of oops is: %p\n",&oops);
printf("Contents of oops is: %d\n",oops);
printf("Address of oops plus is:  %p\n",&oops + 0x8);
printf("Contents of oops plus is: %d\n", *(&oops + 0x8));


}
```

The output of the terminal after the program is entered and the compiler has bee run should look like this:



```
>nano oops.c
>gcc oops.c
>ls
address.c  a.out  helloworld.c  oops.c
>./a.out
Address of oops is: 0x7ffdff637dfc
Contents of oops is: 8
Address of oops plus is:  0x7ffdff637e1c
Contents of oops plus is: 32765
>
```

So much like the previous example we declare an integer and display it's address in memory (line 10) and then its value (line 11). Next we few grab a random address that is 0x8 away from oops and display that (line 12) and its contents (line 13) to the screen with the puzzling result of 32765. What is that value? What uses it? I'm not sure, the only thing I can say is that this location in memory is 8 bytes higher than the memory address of oops. Let's extend this investigation and try to demonstrate an exploit using this idea of accessing memory you aren't supposed to.

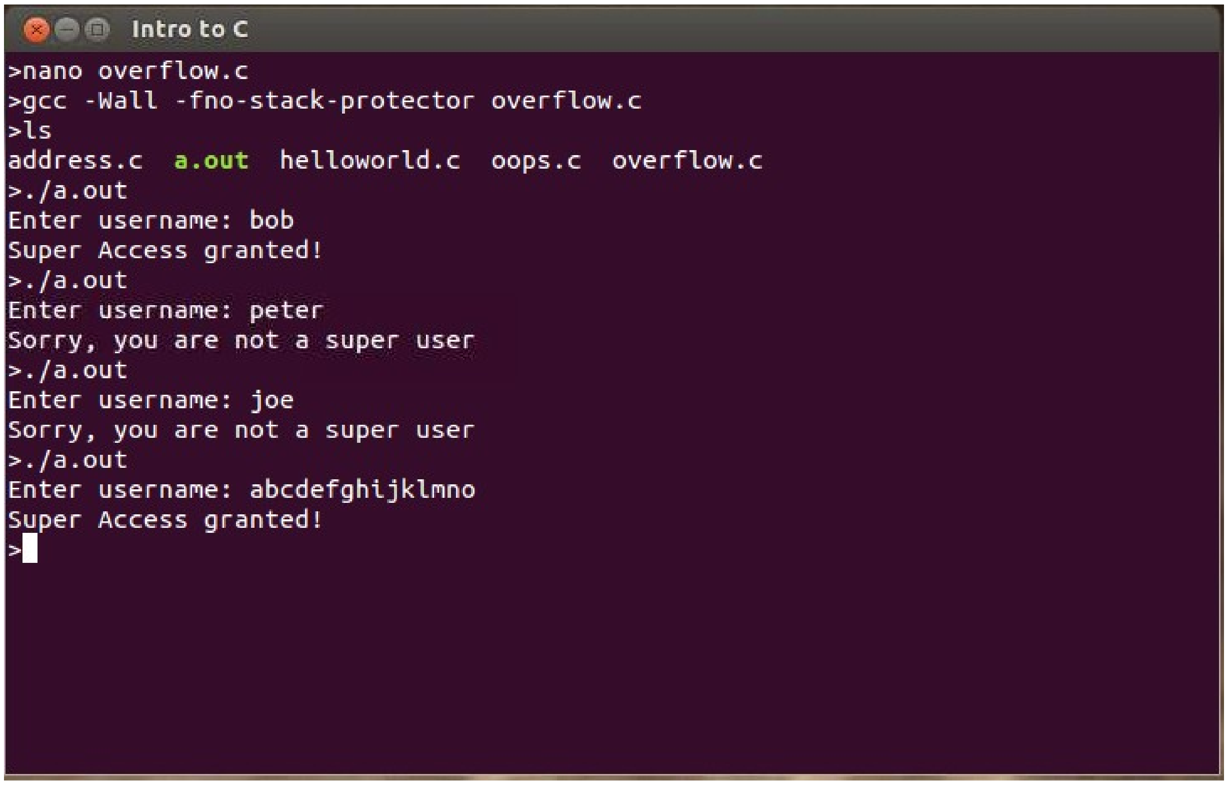
Example 4 Overflow

Try this bit of code into a file named overflow.c

```
/* Buffer Overflow */

#include <stdio.h>
```

```
#include <string.h>

int main()

{


int super_user = 0;

char login[10];


printf("Enter username: ");

gets(login);


  if (strcmp("bob",login) == 0 )

  {

    super_user = 1;

  }


  if(super_user)

  {

    printf("Super Access granted!\n");

  }

  else

  {

    printf("Sorry, you are not a super user\n");

  }


/* Do some other things only a super user can do */
return 0;

}
```

Here is the output of the code writing, save, compile cycle with a few additional runs of the final product. Notice we have to pass some extra information to gcc by way of command line parameters to shut off some built in protections the compiler usually uses.



```
>nano overflow.c
>gcc -Wall -fno-stack-protector overflow.c
>ls
address.c  a.out  helloworld.c  oops.c  overflow.c
>./a.out
Enter username: bob
Super Access granted!
>./a.out
Enter username: peter
Sorry, you are not a super user
>./a.out
Enter username: joe
Sorry, you are not a super user
>./a.out
Enter username: abcdefghijklmno
Super Access granted!
>
```

Here's the line by line of what is introduced here:

Line 3. - We include string.h so that we can use the strcmp call later on.

Line 4. - We declare our main function returns an int or integer value. Used to make the example work as intended.

Line 7.- We declare a character array of length 10. Which is how C for better or worse represents strings.

Line 13. - We use the built in gets function to get input from the keyword up until the user hits enter.

Line 15. - We compare the value entered by the user to the magical super user account name. C compares strings in canonical way. Ie character by character and gives you the difference as an integer value . So two identical strings have a difference of 0. In this case if it checks out super_user is set to 1 or true.

Line 20. - If super_user is true we know the user has special privileges so we let them know about it using a printf statement.

Line 27. - To conclude the main function without compilation error we return an arbitrary integer 0. Much like with the string compare we use 0 here to mean successful, unlike the check to see if super_user where a non zero number means success. Yup another C-ism in action.

This example is a contrived one to demonstrate the ability C has to create something called a buffer overflow. Buffer overflows are essentially exploits where you cram too much data into a series of memory locations and something odd happens as a consequence. This simple program first asks for someone to enter in a user name. If that user name is bob it then grants that user super privileges. However this program is exploitable. The author of the software thought that a user name would only be 10 characters long. However if you were an enterprising individual you could find out that if you entered in a user name longer than 10 characters the program tries to cram in all of those characters into 10 spots and it dribbles out into adjacent memory and causes all sorts of problems. In this case it shortcuts the attempt to compare the entered user name to “bob” and set super_user to 1 if that is the case. To beat this example even more over the head the gcc line used to compile the program is a bit different then what we have seen so far. In this case we are tell the compiler to explicitly not check against buffer overflows.

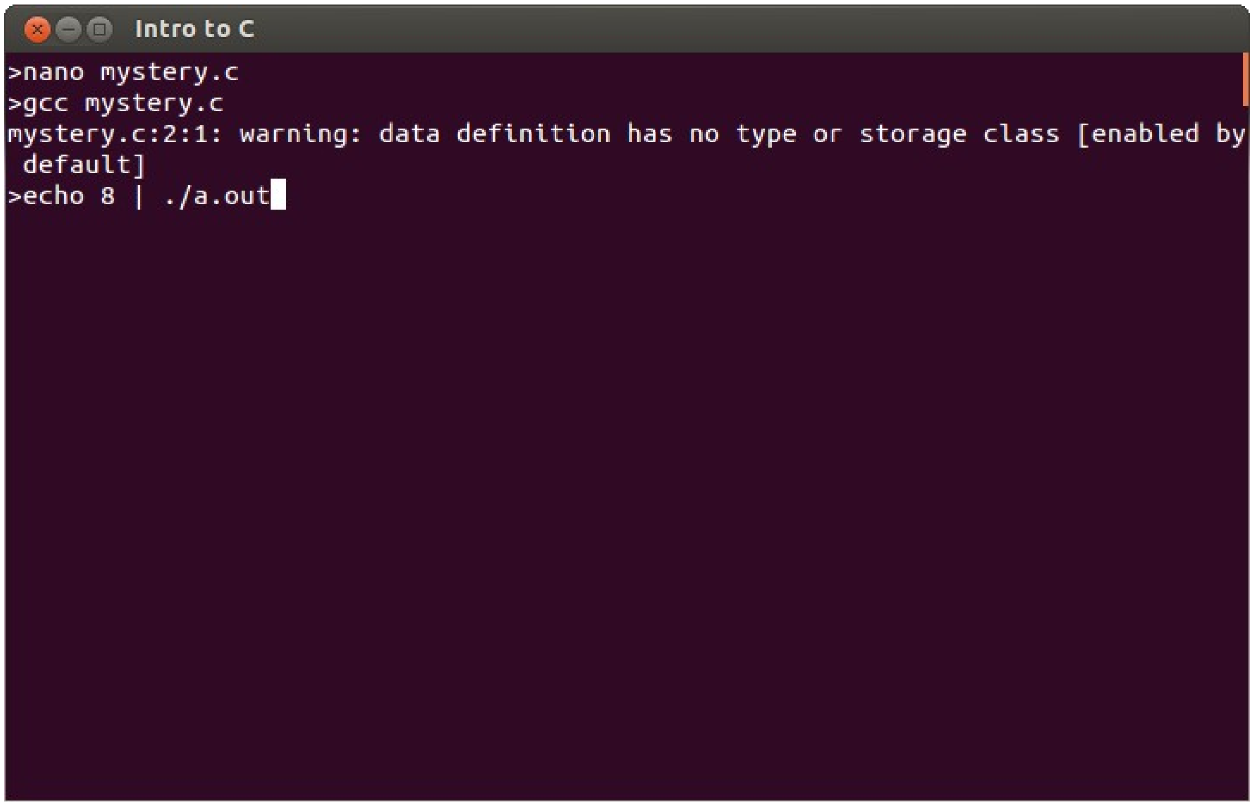
Example 5 – Mystery Code

If C looks confusing, well its' because it is. As demonstrated previously accessing everything in C is done at a very low machine level. Down to the point of looking directly at memory locations. The data structures that you are given are pretty minimal, basically a continuous block of memory you need to very carefully reference. These low level specification create code that works well but the trade off is code that can potentially be totally unreadable. This incongruence is taken to a shocking extreme with a series of events C coders participate in. For example the Underhanded C Contest is a competition where the entrants have to create a piece of code that seems benign but actually performs some dubious function. In a previous year's contest the goal was to create software to emulate an airline luggage processing system that must secretly reroute certain luggage if a particular free text note is applied to the record. The code also had to be written in a way that someone looking at it couldn't divine its purpose.

Another gem is the International Obfuscated C Code Contest. Take for instance this following code block which is a slight modification of some code that was submitted in 1990. Enter in the following in a file called mystery.c

```
#include<stdio.h>
v,i,j,k,l,s,a[99];
main()
{
        for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&(!k&&!!
printf(2+"\n\n%c"-(!l<<!j)," #Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-
i+j&&v+i-j))&&!(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i])
                ;
}
```

Without a clear hint and some diagrams and polite description it would most likely be impossible to infer the meaning of this code. The organizers of the contest have provided some hints on how to run this and what the results should look like. Here is the code entry and compiling step:

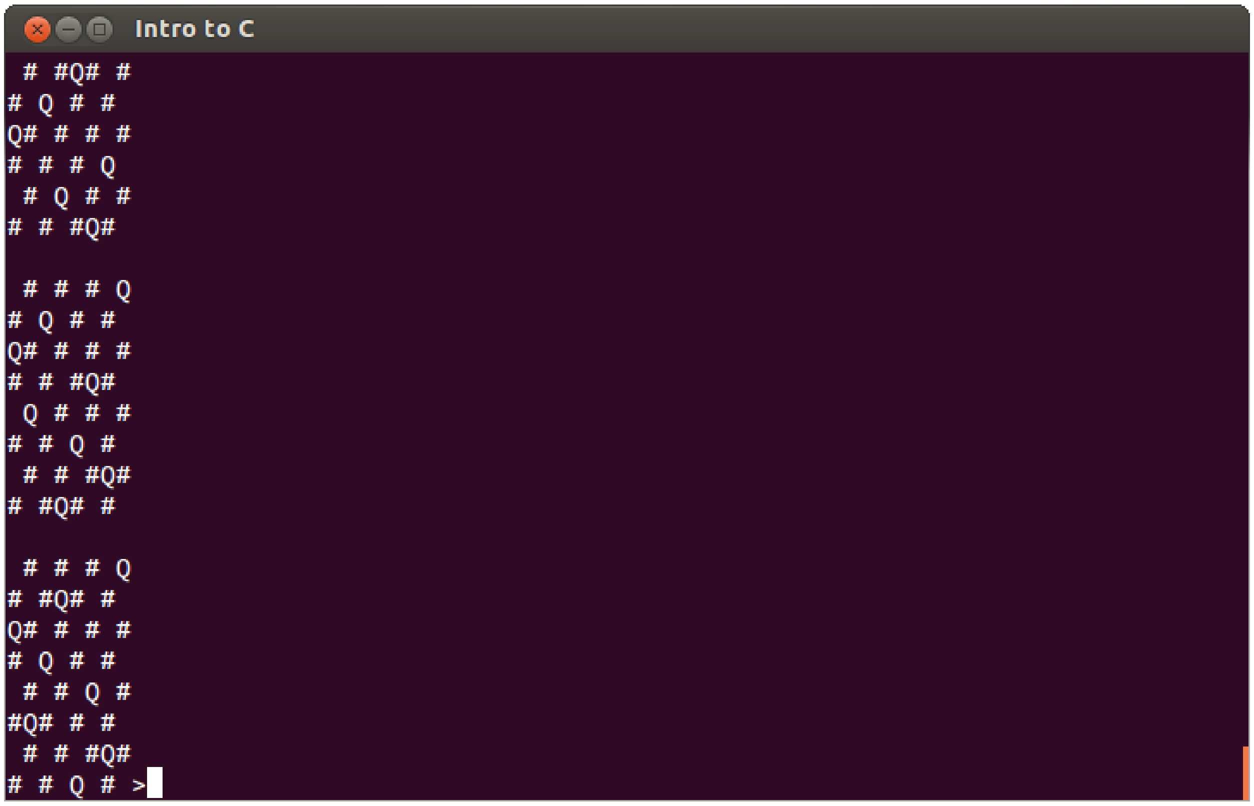


```
>nano mystery.c
>gcc mystery.c
mystery.c:2:1: warning: data definition has no type or storage class [enabled by
 default]
>echo 8 | ./a.out
```

You will probably get a bit of a error message but it is safe to ignore. Before moving to the description and final product of this, let us look at the last line of the previous screen capture:

```
echo 8 | ./a.out
```

Here is another perfect example of how tightly coupled C is with Unix. If we read this line left to right. `echo` is the name of the Unix utility that echos to the screen whatever follows it. `8` is what you expect it to be, the numeral 8. Next is the | symbol. In Unix this is a command line argument that essentially says take the output of the program on the left (a numeral `8` presented to the screen) and uses it for the input of the next program, in our case the mysterious code we just compiled. This process is called piping. You can pipe the output of pretty much any Unix command into any other Unix command, which makes it possible to do very complicated operations on the computer in short order. To return to the example have a look at the output it generates. Before reading on to see the description, can you determine what the code is doing:



```
 # #Q# #
# Q # #
Q# # # #
# # # Q
 # Q # #
# # #Q#

 # # # Q
# Q # #
Q# # # #
# # #Q#
 Q # # #
# # Q #
 # # #Q#
# #Q# #

 # # # Q
# #Q# #
Q# # # #
# Q # #
 # # Q #
#Q# # #
 # # #Q#
# # Q # >
```

In short this code is displaying all the solutions to the 8 Queens problem. This is where you take a chess board and try to place 8 Queen characters on the squares so that none of the Queens are attacking one another. Yeah I didn't get that either from looking at the code. In our emoticon style diagram a space represents a white square, a # is a dark square, and the Q is a Queen piece. The good thing about this code is that it finds the solutions for Queens problem for whatever number you feed into it. You simply change the number from 8 to something else. If you notice when you hit the enter button on the last line a large string of text zipped past the screen. There is actually a Unix command called `more` which will only output a screen full of text before stopping and requiring the user to hit the space bar before showing another screen full of text. Knowing what you know about the | operator, how can you modify our line from above to pipe the output from our compiled code into the more command? The answer to this is left as an exercise for the reader.

This has been a whirlwind introduction to C that has emphasized how the language is used in a Linux based system and has been meant to demonstrate the primary use cases you'd implement with the language. The takeaway is that C is great at hardware programming an situations where you want a really low level interaction with the computer. You probably wouldn't want to use the language for any HTML work. It just isn't a plausible tool for something like that. C takes some work to understand, but the effort invested will pay dividends as you'll be able to program well (because C forces you to write everything with simple data structures) and you'll also develop a strong understanding of the underlying operating system and where and how things are allocated in memory. These are the major tenets of C and as time has demonstrated are enduring qualities that keeps the language relevant to this today. Put in other words, the more time you spend learning C the more easy every other language will be to understand.

Bibliography

Bhaskar, K. "C - Past, Present, and Future - A Perspective." *Resonance: Journal of Science Education* 17, no. 8 (August 2012): 748–58. doi:10.1007/s12045-012-0085-9.

Bhatt, Pramod. “UNIX: Genesis and Design Features.” *Resonance: Journal of Science Education* 17, no. 8 (August 2012): 727–47. doi:10.1007/s12045-012-0084-x.

Daintith, John, and Edmund Wright. *Posix*. Oxford University Press, 2008.

Dowling, Clay. “Using C for CGI Programming.” *Linux Journal*, no. 132 (April 2005): 84–89.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall Software Series. Englewood Cliffs, N.J: Prentice-Hall, 1978.

Klemens, Ben. *21st Century C: C Tips from the New School*. Second edition. Sebastopol, CA: O’Reilly Media, Inc, 2014. Print.

Horton, Ivor, *Beginning C from Novice to Professional.* Springer E-Books, 2006. http://dx.doi.org/10.1007/978-1-4302-0243-1.

Isaak, James. “The History of Posix: A Study in the Standards Process.” *Computer*, no. 7 (1990): 89.

Oliphant, Zan. “Programming CGI in C.” *PC Magazine* 16, no. 5 (March 4, 1997): 235.

Purdum, Jack J., *Beginning C for Arduino.* Springer E-Books, 2006 http://dx.doi.org/10.1007/978-1-4302-4777-7.

Rajaraman, V. “Dennis M Ritchie.” *Resonance: Journal of Science Education* 17, no. 8 (August 2012): 721–23. doi:10.1007/s12045-012-0083-y.

Vine, Michael A. *C Programming for the Absolute Beginner the Fun Way to Learn Programming*. Cincinnati, Ohio: Premier Press, 2002.

Recommended Resources

- CygWin - https://www.cygwin.com/
- cryptlib - https://www.cs.auckland.ac.nz/~pgut001/cryptlib/
- OpenSSL - https://www.openssl.org/
- The LibraryBox Project - http://librarybox.us/
- Underhanded C Contest - http://www.underhanded-c.org/
- 7th International Obfuscated C Code Contest. http://www.ioccc.org/years.html#1990
- “You Can’t Dig Upwards.” An insightful plea on learning C as your first programming language http://www.evanmiller.org/you-cant-dig-upwards.html
- “Death To C”. A look at why bad code in C is causing problems http://techcrunch.com/2015/05/02/and-c-plus-plus-too/
- “The GTK+ Project.” http://www.gtk.org/
- OpenWrt Wiki http://wiki.openwrt.org/
- Zend Engine and PHP https://www.zend.com/en/community/php