Writing Portable and Safe C/C++ Programs

C Programming for Engineers

Nick Urbanik nicku@nicku.org

This document Licensed under GPL-see slide 6

2005 September

Outline

Contents

1	Portable Programming		
	1.1	What is a "portable" program?]
	1.2	Standard Library Functions	2
	1.3	Size of Data	2
	1.4	Order and Arrangement of Data	3
2	Safe Programming		
	2.1	What is a secure program?	2
	2.2	Main sources of problems	4
	2.3	Avoiding Buffer Overflows	
	2.4	Avoiding writing to uninitialised pointers	
	2.5		
3	Refe	prences	•

1 Portable Programming

1.1 What is a "portable" program?

What is a "portable" program?

• A portable program can be compiled and will run successfully on many different *compilers*, *operating systems* and *hardware platforms* with *little or no change* to the source code

- Changes will be easier to make to enable this program to run on a new platform
 - compared with a program that was not written with care about portability.

Way to reduce portability problems

- Avoid proprietary or non-standard libraries
- Avoid assumptions about the size of data
 - Use the definitions in limits.h and math.h
- Avoid assumptions about the order and arrangement of data
 - Some machines are *big-endian*, others (such as the PC) are *little endian*
- Put architecture-dependent code into a separate module
- Be careful when you specify *file names*
- Use the "binary" type when you *read/write binary files*, even if it is not required on your platform
 - otherwise the compiler will treat your file as a text file and corrupt it

1.2 Standard Library Functions

Standard Library Functions

- I see lots of you using the conio.h header.
- Please use this *only* when absolutely necessary!
- Use standard library functions wherever you possibly can.
- Avoid using library functions that start with an underscore, such as _rotr() provided by the Borland 3.1 compiler, and declared in the stdlib.h "standard" header file!
 :-)

1.3 Size of Data

Size of Data

- Many homework exercises assumed that integers are 16 bits long...
- ... this code will *not* run correctly under a 32-bit operating system such as Windows XP or Linux!
- Use **sizeof** and the constant *CHAR_BITS* defined in **#include** < *limits.h*> if you need bit-level information about the size of data on your platform.

Size of Data: Examples

• Code with *many* assumptions about data size:

```
void bin1( unsigned int d )
{
    for ( int i = 0; i < 16; i++ ) {
        int a = ( ( d & 32768 ) == 0 ) ? 0 : 1;
        cout «< a;
        d «= 1;
    }
}</pre>
```

• Code with fewer assumptions about data size:

```
#include #include limits.h>

const int numbits = CHAR_BIT * sizeof( int );

void printbinary( int n )
{
      for ( int i = numbits - 1; i >= 0; --i ) {
            cout « ( ( 1 « i ) & n ? "1" : "0" );
      }
}
```

Exercise: two minutes

- Form a two-person group with the person next to you
- Discuss ways you could make *your own code that you have given for homework more portable*.
- Be ready to *report back* to the class the ways your group could improve the portability of your code.

1.4 Order and Arrangement of Data

Order and Arrangement of Data

- Suppose on some computer
 - a long is 32 bits in size
 - the address of the long variable is 0xb0123456
 - we put the long value 0x12345678 in this variable.

- What byte is stored at 0x12345678?
 - is it 0x12 or 0x78?
- Answer: "it depends"
- On a big-endian machine, such as a Motorola Dragonball processor, the answer is 0x78
- On a *little-endian* machine, such as a PC, the answer is 0x12
- Do not write code that assumes either.

2 Safe Programming

2.1 What is a secure program?

What is a "safe" program?

- A secure program cannot be easily exploited by a malicious person to gain privileges that they should not have
- A secure program will run more *reliably*
 - Not "sometimes run okay, other times it crashes"
- Symptoms of possible security problems include:
 - occasionally terminates with a "segmentation fault" or "protection error"
 - data occasionally appears with unrecognisable garbage appended
 - changing one data item causes another unrelated data item to change

2.2 Main sources of problems

Main sources of problems

- Writing *past the end of arrays* on the stack
 - Exploited by crackers as a technique described in Smashing The Stack For Fun And Profit by Elias Levy (aka Aleph One) at http://www.insecure.org/stf/s and http://www.phrack.org/show.php?p=49&a=14
- writing to *uninitialised pointers*
- *memory allocation* errors:
 - allocating memory without freeing it ("memory leak")
 - freeing memory twice ("double free")

2.3 Avoiding Buffer Overflows

Avoiding Buffer Overflows

- When reading strings into arrays, always use techniques that limit the data read into the string and make sure it is null terminated.
- With iostreams:
 - use the *istream*::*getline*() method to read input lines, limiting the number of bytes read to the length of the buffer
 - or you can use the setw() iostream manipulator to limit characters read (#include <ioi
- *Never* use the *gets*() library function
- use stncpy() rather than strcpy(), use strncat() rather than strcat(), ...
- Simply make sure that there is *no possibility* of writing past the end of an array.

2.4 Avoiding writing to uninitialised pointers

Avoiding writing to uninitialised pointers

- Before you use a pointer, it has some uninitialised value, and points to some *random location*
- You must have the pointer *point* somewhere to memory that you own *before* you write to the location.
- How? Either:
 - make the pointer point to an existing variable, or
 - allocate some memory dynamically (with the C++ new operator or the malloc() library function)

2.5 Avoiding memory allocation problems

Avoiding memory allocation problems

- It is up to you to remember where you allocated memory
- For each piece of memory you allocate, it will not be freed up till either you free it up, or the program terminates.
- If the program will run a long time, and will make many allocations, then you need to be like an accountant: you have to free it up.

3 References

References

References

- [1] Elias Levy, aka Aleph One. *Smashing The Stack For Fun And Profit.* Phrack No. 49, 8 November 1996. http://www.phrack.org/show.php?p=49&a=14. http://destroy.net/machines/security/.
- [2] Steve Oualline. Practical C Programming. O'Reilly, 1993.
- [3] Paul Davies. *The Indispensable Guide to C with Engineering Applications* Addison-Wesley, 1995.
- [4] Brian W. Kernighan and Desnnis M. Ritchie. *The C Programming Language*. Prentice Hall. 1988.

License covering this document

Copyright © 2005 Nick Urbanik <nicku@nicku.org>

You can redistribute modified or unmodified copies of this document provided that this copyright notice and this permission notice are preserved on all copies under the terms of the GNU General Public License as published by the Free Software Foundation — either version 2 of the License or (at your option) any later version.

5

6