

C++ TUTORIAL - EMBEDDED SYSTEMS PROGRAMMING - 2017



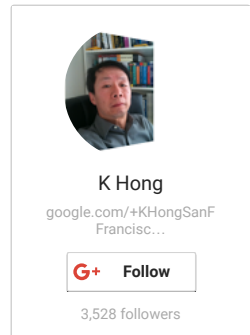
(<http://www.addthis.com/bookmark.php?v=250&username=khhong7>)

bogotobogo.com site search:

bogotobogo.com site search:

Embedded Systems

When we talk about **embedded systems** programming, in general, it's about writing programs for **gadgets**.



Ph.D. / Golden Gate Ave, San Francisco / Seoul National Univ / Carnegie Mellon / UC Berkeley / DevOps / Deep Learning / Visualization

Sponsor Open Source development activities and free contents for everyone.



Thank you.

- K Hong (http://bogotobogo.com/about_us.php)



Sponsor Open Source development activities and free contents for everyone.



Thank you.

- K Hong (http://bogotobogo.com/about_us.php)

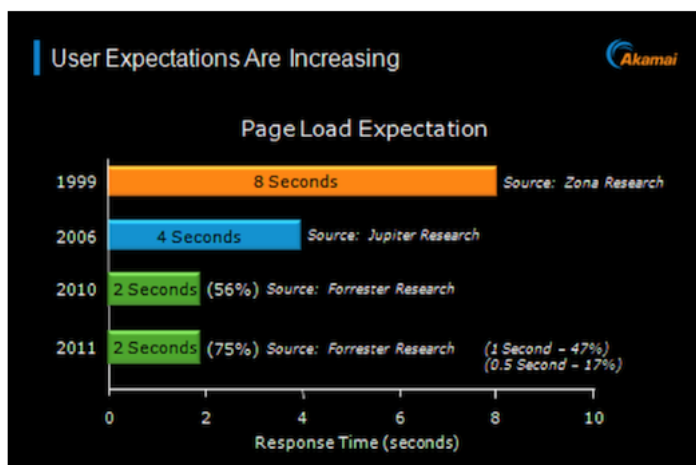


bogotobogo.com site search:

bogotobogo.com site search:

Gadget with a brain is the embedded system. Whether the brain is a **microcontroller** or a **digital signal processor (DSP)**, gadgets have some interactions between hardware and software designed to perform one or a few **dedicated functions**, often with **real-time computing** constraints.

Usually, embedded systems are resource constrained compared to the desktop PC. Embedded systems, typically, have limited memory, small or no hard drives, and in some cases with no external network connectivity.



Picture: <https://blogs.akamai.com> - The Performance Arms Race (<https://blogs.akamai.com/>)

C++ Tutorials

C++ Home
(/cplusplus/cpptut.php)

Algorithms & Data Structures in C++ ...
(/Algorithms/algorithms.php)

Application (UI) - using Windows Forms (Visual Studio 2013/2012)
(/cplusplus/application_visual_studio.php)

auto_ptr
(/cplusplus/autoptr.php)

Binary Tree Example Code
(/cplusplus/binarytree.php)

Blackjack with Qt
(/cplusplus/blackjackQT.php)

Boost - shared_ptr, weak_ptr, mpl, lambda, etc.
(/cplusplus/boost.php)

Boost.Asio (Socket Programming - Asynchronous TCP/IP)...
(/cplusplus/Boost/boost_AsyncIO.php)

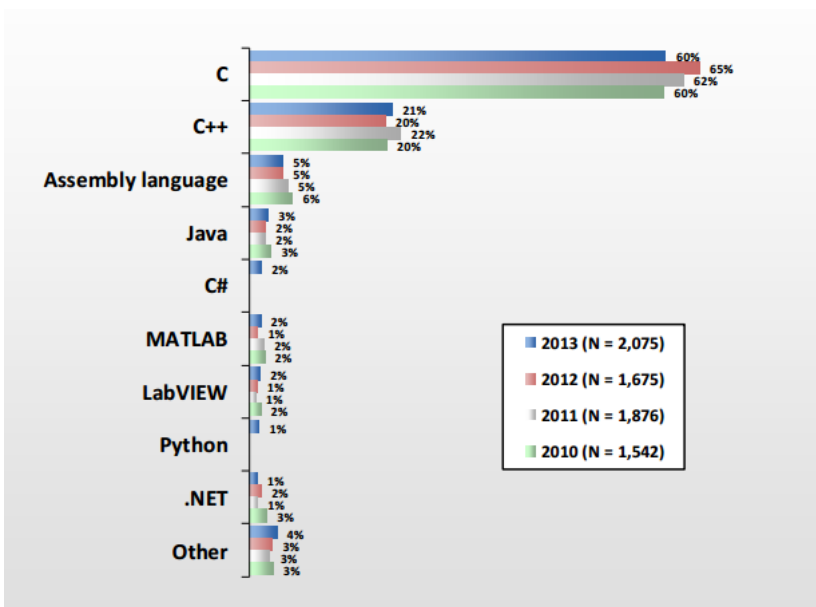
Classes and Structs
(/cplusplus/class.php)

Constructor
(/cplusplus/constructor.php)

C++11(C++0x): rvalue references, move constructor, and lambda, etc.
(/cplusplus/cplusplus11.php)

C++ API Testing
(/cplusplus/cpptesting.php)

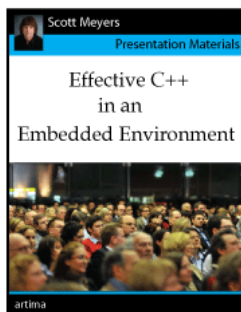
C++ Keywords - const, volatile, etc.



Picture: Programming languages for embedded systems

(http://images.content.ubmtechelectronics.com/Web/UBMTechElectronics/%7Ba7a91f0e-87c0-4a6d-b861-d4147707f831%7D_2013EmbeddedMarketStudyb.pdf)

Presentation Materials: Effective C++ in an Embedded Environment (pdf) by Scott Meyers, 2012

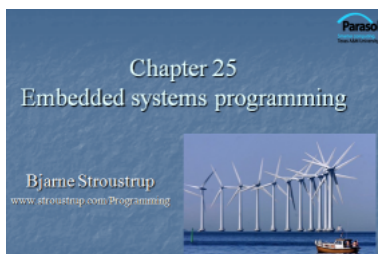


Last revised
October 4, 2012
320 pages
([View a free sample](#))

(</cplusplus/files/effCppEmbNotesSample.pdf>)

http://www.artima.com/shop/effective_cpp_in_an_embedded_environment.

(http://www.artima.com/shop/effective_cpp_in_an_embedded_environment)



etc.

(/cplusplus/cplusplus_keywords.i)

Debugging Crash & Memory Leak

(</cplusplus/CppCrashDebuggingI>)

Design Patterns in C++ ...

(</DesignPatterns/introduction.ph>)

Dynamic Cast Operator

(/cplusplus/dynamic_cast.php)

Eclipse CDT / JNI (Java Native Interface) / MinGW

(/cplusplus/eclipse_CDT_JNI_MinG)

Embedded Systems

Programming I - Introduction

(</cplusplus/embeddedSystemsPr>)

Embedded Systems

Programming II - gcc ARM

Toolchain and Simple Code on

Ubuntu and Fedora

(</cplusplus/embeddedSystemsPr>)

Embedded Systems

Programming III - Eclipse CDT

Plugin for gcc ARM Toolchain

(</cplusplus/embeddedSystemsPr>)

Exceptions

(</cplusplus/exceptions.php>)

Friend Functions and Friend Classes

(</cplusplus/friendclass.php>)

fstream: input & output

(/cplusplus/fstream_input_outpu)

Function Overloading

(/cplusplus/function_overloading)

Functors (Function Objects) I - Introduction

(/cplusplus/functor_function_obji)

Functors (Function Objects) II -

Converting function to functor

(/cplusplus/functor_function_obji)

Functors (Function Objects) -

Presentation Materials: Embedded Systems Programming (ppt)

by Bjarne Stroustrup (http://www.bogotobogo.com/cplusplus/files/embed/25_embedded.ppt)

or get it from <http://www.stroustrup.com/Programming/> (<http://www.stroustrup.com/Programming/>).

Why Linux?

According to LinuxDevices.com (<http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Snapshot-of-the-embedded-Linux-market-April-2007>), Linux has emerged as the dominant OS

for embedded systems, and it will reach 70% of the market by 2012.

Go to Embedded Linux.

Issues for Embedded Systems Programming

Here are some characteristics of embedded systems, and few systems suffer all of these constraints.

1. Reliability - failure is very expensive.
2. Limited Resources - A resource is something of which a machine has only a limited supply. As an embedded systems programmer, we get it through some explicit action such as "acquire" or "allocate" and return it through "release", "free", or "deallocate" to the system. It could be memory, file handles, network connection or communication channels such as sockets, and locks. Or it could be processor cycles, power.
3. Real-time Response.
4. A system may be Running Forever.

The characteristics of embedded systems affect the embedded systems programming:

1. Correctness - producing the results at the right time, in the right order, and using only an acceptable set of resources.
2. Fault tolerance
3. No downtime.
4. Real-time constraints.
 1. hard real time: if a system's response must occur before a deadline
 2. soft real time: same as the hard real time but an occasional miss can be affordable
5. Predictability - In programming embedded systems, predictability usually means the predictability of the time it takes for certain operation. So, operations that are not guarantee a response within a given time, should not be used. For example, a linear search of a list is an unpredictable operation because its number of elements is unknown.

General
(/cplusplus/functors.php)

Git and GitHub Express...
(/cplusplus/Git/Git_GitHub_Expr...

GTest (Google Unit Test) with
Visual Studio 2012
(/cplusplus/google_unit_test_gtes

Inheritance & Virtual
Inheritance (multiple
inheritance)
(/cplusplus/multipleinheritance.p

Libraries - Static, Shared
(Dynamic)
(/cplusplus/libraries.php)

Linked List Basics
(/cplusplus/linked_list_basics.php

Linked List Examples
(/cplusplus/linkedlist.php)

make & CMake
(/cplusplus/make.php)

make (gnu)
(/cplusplus/gnumake.php)

Memory Allocation
(/cplusplus/memoryallocation.ph

Multi-Threaded Programming -
Terminology - Semaphore,
Mutex, Priority Inversion etc.
(/cplusplus/multithreaded.php)

Multi-Threaded Programming II
- Native Thread for Win32 (A)
(/cplusplus/multithreading_win3:

Multi-Threaded Programming II
- Native Thread for Win32 (B)
(/cplusplus/multithreading_win3:

Multi-Threaded Programming II
- Native Thread for Win32 (C)
(/cplusplus/multithreading_win3:

Multi-Threaded Programming II

6. Concurrency.

So, when we do embedded systems programming, we should be aware of the environment and its use. In other words, domain knowledge is essential for designing and implementing a system with a good protection against errors.

1. We're dealing with specialized features of RTOS.
2. We're using a **non-hosted environment**. In other words, we're using a language right on top of hardware without any protection or help from the traditional OSs.
3. We're dealing with device drivers/hardware device APIs.

Real-time Applications - Predictability

Following features of C++ language are not predictable:

1. **new** and **delete**

1. **Static** memory poses no special problem in embedded systems programming since all is taken care of before the program starts to run and long before a system is deployed.
2. **Stack** memory can be a problem because it is possible to use too much of it. One way is to avoid **recursive** functions and stick to **iterative** implementation.
3. **Dynamic** memory allocation is usually banned or restricted. The **new** and **malloc()** are either banned or using them is restricted to a startup period, and **delete** is banned because of the **predictability** and **fragmentation**.

1. **predictability - allocation delays**

The time it takes to find a free chunk of memory of a specific size depends on what's been already allocated.

2. **fragmentation**

After several memory allocations, fragments (hole) may take up much of the memory. However, there are two data structures that are particularly useful for predictable memory allocations: **stacks**, **pools**, and **global objects** :

1. **Pool** is a data structure from which we can allocate object of a given type and later deallocate such object. A pool contains a maximum number of objects; that number is specified when the pool is created. So, it gives us constant time operations without any fragmentation.
2. **Stack** is a data structure from which we can allocate chunks of memory and deallocate the last allocated chunk. Stacks grow and shrink only at the top without any fragmentation, and it guarantees constant time operation.
3. **global objects** can be allocated at startup time so that set can set aside a fixed amount of memory.
4. C++ standard library containers such as **vector**, **map**, etc. and the standard **string** are not to be used because they indirectly use **new**.

- C++ Thread for Win32
(/cplusplus/multithreading_win32)

Multi-Threaded Programming
III - C/C++ Class Thread for
Pthreads
(/cplusplus/multithreading_pthre

MultiThreading/Parallel
Programming - IPC
(/cplusplus/multithreading_ipc.pl

Multi-Threaded Programming
with C++11 Part A (start, join(),
detach(), and ownership)
(/cplusplus/multithreaded4_cplu:

Multi-Threaded Programming
with C++11 Part B (Sharing
Data - mutex, and race
conditions, and deadlock)
(/cplusplus/multithreaded4_cplu:

Multithread Debugging
(/cplusplus/multithreadedDebug

Object Returning
(/cplusplus/object_returning.php

Object Slicing and Virtual Table
(/cplusplus/slicing.php)

OpenCV with C++
(/cplusplus/opencv.php)

Operator Overloading I
(/cplusplus/operatoroverloading.

Operator Overloading II - self
assignment
(/cplusplus/operator_oveloding.

Pass by Value vs. Pass by
Reference
(/cplusplus/valuevsreference.php

Pointers
(/cplusplus/pointers.php)

Pointers II - void pointers &
arrays
(/cplusplus/pointers2_voidpointe

Pointers III - pointer to function

2. Exceptions

How can we catch **all** exceptions and **how long** it will take to find a matching **catch**.

The **throw** is typically banned in hard real-time applications. Instead, we may rely on return codes to do error handling.

Reliability

Avoid language features and programming techniques that have proved error-prone. Pointers!

1. Explicit conversions which are unchecked and unsafe - avoid them.

1. unchecked conversion

Embedded systems often require a programmer to access a specific memory location (0xffa2):

```
class Driver;
Driver *ptr = reinterpret_cast<Driver*>(0xffa2);
```

In this low-level programming, we should be aware that the correspondence between a hardware resource (register's address) and a pointer to the software that manipulates the hardware resource is brittle. Though the **reinterpret_cast** from an **int** to a pointer type is crucial link for connections between an application and its hardware resources, we should not expect a code using (unchecked) `reinterpret_cast` to be portable.

2. Passing pointers to array elements - An array is often passed to a function as a pointer to an element. Therefore, they lose their size, so that the receiving function cannot directly tell how many elements are pointed to. This is a cause of many bugs.

RTOS (real-time operating system)

Realtime applications are those that need to respond in a timely fashion to input. Frequently, such input comes from an external sensor or a specialized input device, and output takes the form of controlling some external hardware.

Although many realtime applications require rapid responses to input, the defining factor is that **the response is guaranteed to be delivered within a certain deadline time after the triggering event**.

The provision of realtime responsiveness, especially where short response times are demanded, requires support from the underlying operating system.

However, most OS does not natively provide such support because the requirements of realtime responsiveness can conflict with the requirements of multiuser timesharing operating systems.

& multi-dimensional arrays
(/cplusplus/pointers3_function_n

Preprocessor - Macro
(/cplusplus/preprocessor_macro.

Private Inheritance
(/cplusplus/private_inheritance.p

Python & C++ with SIP
(/python/python_cpp_sip.php)

(Pseudo)-random numbers in C++
(/cplusplus/RandomNumbers.ph

References for Built-in Types
(/cplusplus/references.php)

Socket - Server & Client
(/cplusplus/sockets_server_client

Socket - Server & Client with Qt (Asynchronous / Multithreading / ThreadPool etc.)
(/cplusplus/sockets_server_client

Stack Unwinding
(/cplusplus/stackunwinding.php)

Standard Template Library (STL) I - Vector & List
(/cplusplus/stl_vector_list.php)

Standard Template Library (STL) II - Maps
(/cplusplus/stl2_map.php)

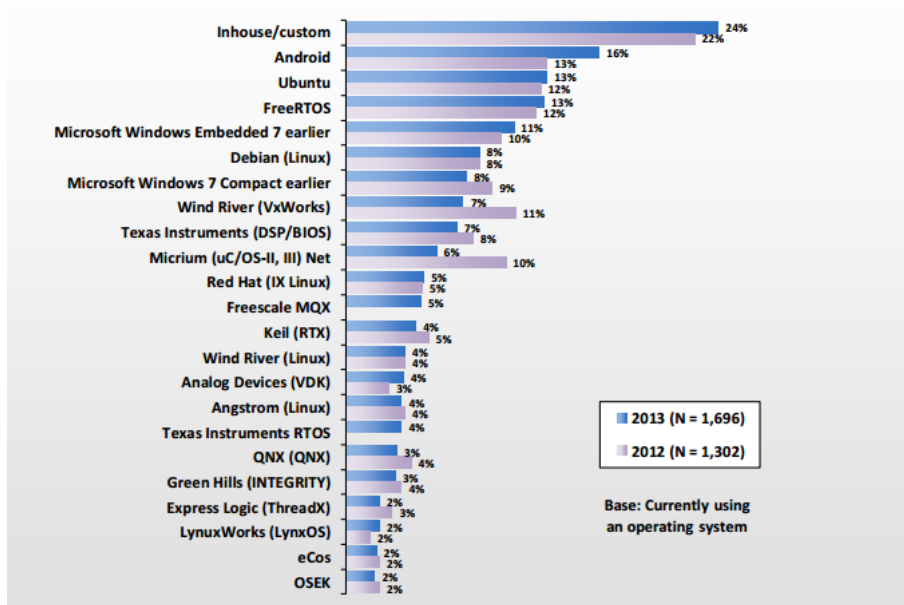
Standard Template Library (STL) II - unordered_map
(/cplusplus/stl2_unorderd_map_c

Standard Template Library (STL) II - Sets
(/cplusplus/stl2B_set.php)

Standard Template Library (STL) III - Iterators
(/cplusplus/stl3_iterators.php)

Standard Template Library (STL) IV - Algorithms
(/cplusplus/stl4_algorithms.php)

Realtime variants of Linux have been created, and recent Linux kernels are moving toward full native support for realtime applications.



Picture source: OS for embedded systems - Embedded Market Study, 2013
(http://www.bogotobogo.com/cplusplus/files/embed/EmbeddedMarketStudy_2013.pdf)

An **RTOS** is an operating system designed to meet strict deadlines which associated with tasks. In RTOS, therefore, missing the deadline can cause undesired or even catastrophic outcome.

I/O Latency

I/O type	cycles
L1	3
L2	14
RAM	250
Disk	41,000,000
Network	240,000,000

	L2	RAM	Disk	Network
L1	5	83	13,666,666	80,000,000
L2		18	2,928,571	17,142,857
RAM			164,000	960,000

Standard Template Library (STL) V - Function Objects
(/cplusplus/stl5_function_objects)

Static Variables and Static Class Members
(/cplusplus/statics.php)

String (/cplusplus/string.php)

String II - stringstream etc.
(/cplusplus/string2.php)

Taste of Assembly
(/cplusplus/assembly.php)

Templates
(/cplusplus/templates.php)

Template Specialization
(/cplusplus/template_specialization.php)

Template Specialization - Traits
(/cplusplus/template_specialization_traits.php)

Template Implementation & Compiler (.h or .cpp?)
(/cplusplus/template_declaration_and_implementation.php)

The this Pointer
(/cplusplus/this_pointer.php)

Type Cast Operators
(/cplusplus/typecast.php)

Upcasting and Downcasting
(/cplusplus/upcasting_downcasting.php)

Virtual Destructor & boost::shared_ptr
(/cplusplus/virtual_destructors_shared_ptr.php)

Virtual Functions
(/cplusplus/virtualfunctions.php)

Programming Questions and Solutions ↓

Strings and Arrays
(/cplusplus/quiz_strings_arrays.php)

Linked List

Disk				6
------	--	--	--	---

The followings are the characteristics of RTOS:

1. Context switching latency:

Context switch latency is the time from one context switching to another and it should be short. In other words, the time taken while saving the context of current task and then switching over to another task should be short. In general, switching context involved saving the CPU's registers and loading a new state, flushing the caches, and changing the virtual memory mapping. Context switch latency is highly architecture dependent and different hardware may get different results.

2. Interrupt latency:

Interrupt latency is the time from interrupt generation until the interrupt service routine starts executing.

Factors that affect interrupt latency include the processor architecture, the processor clock speed, the particular OS employed, and the type of interrupt controller used.

Minimum interrupt latency depends mainly on the configuration of the interrupt controller, which combines interrupts onto processor lines, and assigns priority levels (visit Priority Inversion (http://www.bogotobogo.com/cplusplus/multithreaded.php#priority_inversion)) to the interrupts.

Maximum interrupt latency depends mainly on the OS.

For more on Interrupt and Interrupt Latency, please visit my another page Interrupt & Interrupt Latency

(http://www.bogotobogo.com/Embedded/hardware_interrupt_software_interrupt_latency_irq_vs_fiq.php)

3. Dispatch latency:

The time between when a thread is scheduled and when it begins to execute. Theoretically, in a preemptive OS the dispatch latency for a high-priority thread should be very low. However, in practice preemptive OSs are non-preemptive at times; for example, while running an interrupt handler. The duration of the longest possible non-preemptive interval is said to be the worst-case dispatch latency of an OS.

4. Reliable and time bound inter process mechanisms should be in place for processes to communicate with each other in a timely manner.

5. Multitasking and task preemption:

An RTOS should have support for multitasking and task preemption. Preemption means to switch from a currently executing task to a high priority task ready and waiting to be executed.

6. Kernel preemption:

Most modern systems have preemptive kernels, designed to permit tasks to be preempted even when in kernel mode.

The bright side of the preemptive kernel is that sys-calls do not block the entire system.

However, it introduces more complexity to the kernel code, having to handle more end-cases, perform more fine grained locking or use lock-less structures and algorithms.

7. Note: Preemptive:

Preemptive means that the rules governing which processes receive use of the CPU and for how long are determined by the **kernel process scheduler**.

(/cplusplus/quiz_linkedlist.php)

Recursion

(/cplusplus/quiz_recursion.php)

Bit Manipulation

(/cplusplus/quiz_bit_manipulation.php)

Small Programs (string, memory functions etc.)

(/cplusplus/smallprograms.php)

Math & Probability

(/cplusplus/quiz_math_probability.php)

Multithreading

(/cplusplus/quiz_multithreading.php)

140 Questions by Google

(/cplusplus/google_interview_questions.php)

Qt 5 EXPRESS...

(/Qt/Qt5_Creating_QtQuick2_QMLApplication.php)

Win32 DLL ...

(/Win32API/Win32API_DLL.php)

Articles On C++

(/cplusplus/cppNews.php)

What's new in C++11...

(/cplusplus/C11/C11_initializer_lists.php)

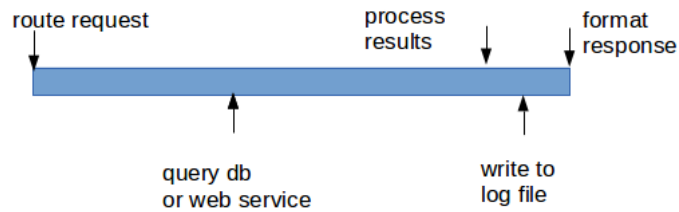
C++11 Threads EXPRESS...

(/cplusplus/C11/1_C11_creating_threads.php)

OpenCV...

(/OpenCV/opencv_3_tutorial_image_operations.php)

*Most of the requests are spending time just waiting due to I/Os



Bits

In code meant to be portable, use should use **<limits>** to make sure our assumption about sizes is correct. Here is the list of sizes of the primitive types:

1. **bool** - 1 bit, but takes up a byte
2. **char** - 8 bits
3. **short** - 16 bits
4. **int** - 32 bits, but many embedded systems have 16-bit **ints**
5. **long int** - 32 bits or 64 bits

bitset

```
#include <iostream>
#include <bitset>
#include <iomanip>

int main( )
{
    using namespace std;

    int i;
    while(cin >> i)
        cout << dec << i << "=="
            << hex << "0x" << i << "=="
            << bitset<8*sizeof(int)>(i)<< endl;
    return 0;
}
```

Output is:


```
#include <iostream>
#include <vector>

int main( )
{
    using namespace std;

    const int max = 10;
    vector<int> v(max,77);

    for(int i = 0; i < v.size(); ++i)
        cout << v[i] << endl;
}
```

When we compile it, we get a warning something like this:

```
signed/unsigned mismatch
```

That's because the index **i** is **signed** integer, but **v.size()** is **unsigned** integer. Mixing signed and unsigned could lead to disaster. For instance, the loop variable **i** might overflow. In other words, **v.size()** might be larger than the largest signed **int**. Then, **i** would reach the highest value that could represent a positive integer in a signed **int**. Then, the next **++** couldn't yield the next-highest integer and would instead result in a negative value. The loop would never terminate!

Here, we have two choices:

1. `vector<int>::size_type`

```
for(vector<int>::size_type i = 0; i < v.size(); ++i)
    cout << v[i] << endl;
```

2. `iterator`

```
for(vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << *it << endl;
```

The **size_type** is guaranteed to be unsigned, so the first form has one more bit to play with than the **int** version. That can be significant, but it is still gives only a single bit of range. The loop using iterators has no such limitation.

Here is an advice on arithmetic operation with **unsigned integer** by Bjarne Stroustrup (http://www.bogotobogo.com/cplusplus/files/embed/25_embedded.ppt)

```
"Avoid that when you can
-Try never to use unsigned just to get another bit of precision
-If you need one extra bit, soon, you'll need another
-Don't mix signed and unsigned in an expression
You can't completely avoid unsigned arithmetic
Indexing into standard library containers uses unsigned
(in my opinion, that's a design error)"
```

For more on the issue of signed integer and unsigned integer in the embedded systems programming, please visit the following pages:

1. Two's Complement
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#twoscomplement)
2. Unsigned & signed integers
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#signed_unsigned)
3. Unsigned & signed integers as a loop variable
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#signed_unsigned_loop)

Bit Manipulation

When do we need to manipulate bits?

1. flags as hardware indicators
2. low-level communications - we need to extract from byte streams
3. graphics - we need to compose picture out of several images
4. encryption

Here is an example of extracting information from a short integer.

```
#include <iostream>
#include <bitset>
int main( )
{
    using namespace std;

    const int max = 8;

    short val = 4;
    unsigned char left = val & 0xff;          // leftmost (least significant) byte
    cout << bitset<max>(left) << endl;

    unsigned char right = (val >> 8) & 0xff; // rightmost (most significant) byte
    cout << bitset<max>(right) << endl;

    val = 256;
    right = (val >> 8) & 0xff;
    cout << bitset<max>(right) << endl;

    val = 2*val;
    right = (val >> 8) & 0xff;
    cout << bitset<max>(right) << endl;

    val = -8;
    bool sign_bit = val & 0x8000;           // sign bit
    cout << sign_bit << endl;
}
```

Output is:

```
00000100
00000000
00000001
00000010
1
```

The operations are known as **shift** and **mask**. We **shift** to place the bits we want to consider to the rightmost (least significant) part of the word where they are easy to manipulate. We **mask** using and (&) together with a bit pattern such as 0xff to eliminate the bits we do not want in the result.

Here is another example clearing 1st and 3rd bits:

```

#include <iostream>
using namespace std;

#define MASK (0xF5)

void printBinary(unsigned int n)
{
    int i = 7;
    while(i >= 0) {
        cout << (n >> i & 1);
        i--;
    }
    cout << endl;
}

int main()
{
    unsigned int a = 15;

    cout << "MASK      : ";
    printBinary(MASK);

    cout << "a        : ";
    printBinary(a);

    a &= MASK;
    cout << "a & MASK: ";
    printBinary(a);

    return 0;
}

```

Output:

```

MASK      : 11110101
a         : 00001111
a & MASK: 00000101

```

Example: **enum**, bit set and testing bits:

```

enum BIT_VECTOR
{
    bit7 = 1<<7, bit6 = 1<<6, bit5 = 1<<5, bit4 = 1<<4,
    bit3 = 1<<3, bit2 = 1<<2, bit1 = 1<<1, bit0 = 1
};

int main()
{
    unsigned char val = bit4 | bit2;    // val = 16 + 4 = 20
    val |= bit1;                       // val = 16 + 4 + 2 = 22
    if(val & bit1);                     // bit1 is set? yes!

    unsigned char new_val = val & (bit3 | bit4);    // new_val = 16
    BIT_VECTOR another = BIT_VECTOR(bit3 | bit1);   // 8 + 2 = 10
    return 0;
}

```

Bit Field example:

The declaration of a bit-field has the form inside a structure:

```
struct
{
    type [member_name] : width ;
};
```

When we use the following structure, we end up using just 4 byte rather than 4+4+4+4=16 bytes:

```
#include <iostream>
using namespace std;

struct bitfield {
    unsigned int b1 : 1;      // will use 1 bit
    unsigned int b2 : 1;      // 1 bit
    unsigned int nibble : 4;   // 4 bits
    unsigned int byte : 8;     // 8 bits
} bf;

int main()
{
    cout << sizeof(bf) << endl;    // prints out 4
    return 0;
}
```

For more on bit manipulation, please visit

http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php

(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php) where the following issues are discussed:

1. Bitwise Operations
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#bitwiseoperations)
2. Setting and Clearing a Bit
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#setclear)
3. Displaying an Integer with Bits
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#displaybit)
4. Converting Decimal to Hex
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#dectohex)
5. Bit Field and Struct (http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#bitfield)
6. Bit Field, Struct & Union
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#bitfield_union)
7. The Number of Bits Set in an Integer (Number of Ones)
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#bitcount)
8. The Bit Set Position of an Integer
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#bitsetpos)
9. In-Place Integer Swap with Bit Manipulation
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#swap)

10. The Number of Bits Required to Convert an Integer A to Integer B
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#numberofbits)
11. Swap Odd and Even Bits in an Integer
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#swapevenodd)
12. What $(n \& (n-1) == 0)$ is checking?
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#powerof2)
13. Two's Complement
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#twoscomplement)
14. Unsigned & signed integers
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#signed_unsigned)
15. Unsigned & signed integers as a loop variable
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#signed_unsigned_loop)
16. What does this do? $n \wedge= INT_MIN$
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#INT_MIN_XOR)
17. Flipping n-th bit of an integer
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#bit_flip)
18. Big Endian & Small Endian
(http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php#endian)

For keywords such as **volatile** or **const volatile**, visit C++ Keywords
(http://www.bogotobogo.com/cplusplus/cplusplus_keywords.php#volatile).

Processors of Embedded Systems

From wiki (http://en.wikipedia.org/wiki/Embedded_system)

Firstly, Embedded processors can be broken into two broad categories: ordinary **microprocessors** (**µP**) and **microcontrollers** (**µC**), which have many more peripherals on chip, reducing cost and size. Contrasting to the personal computer and server markets, a fairly large number of basic CPU architectures are used; there are **Von Neumann** as well as various degrees of **Harvard architectures**, **RISC** as well as **non-RISC** and **VLIW**; word lengths vary from **4-bit** to **64-bits** and beyond (mainly in DSP processors) although the most typical remain **8/16-bit**. Most architectures come in a large number of different variants and shapes, many of which are also manufactured by several different companies.

A long but still not exhaustive list of common architectures are: 65816, 65C02, 68HC08, 68HC11, 68k, 8051, **ARM**, AVR, AVR32, Blackfin, C167, Coldfire, COP8, Cortus APS3, eZ8, eZ80, FR-V, H8, HT48, M16C, M32C, MIPS, MSP430, PIC, **PowerPC**, R8C, SHARC, SPARC, ST6, SuperH, TLCS-47, TLCS-870, TLCS-900, Tricore, V850, x86, XE8000, Z80, AsAP etc.

Embedded Software & Tools Market in 2011 by VDC Research

(http://www.vdcresearch.com/market_research/embedded_software/productid=2646)

1. COMs gain traction as time-to-market accelerators for OEMs

By combining COM express modules with off-the-shelf COMs, suppliers are able to offer several different configurations of CPU boards and leverage COMs' interchangeable characteristics. CPU vendors can thus offer a fairly wide range of boards without incurring high design and inventory carrying costs.

2. PC/104 module family under pressure

Although VDC data projects the PC/104 family will experience a single-digit rebound from the low points of the recent recession, vendors will have to commit resources to developing newer strategies in order for this technology to remain viable. Otherwise, the recovery of these architectures is likely to stall or decline in 2011.

3. Asia continues to rise in the development of embedded technology

2011 will see further strengthening of the Asian embedded supplier community as supply chain synergies, R&D; capabilities and fabrication automation increases between upstream and downstream ecosystem partners.

4. China's growth will power MCU market

Continued economic growth in China will drive the country's automotive market and expand the need for MCU (microcontroller unit) technology. Despite reduction in government subsidies, VDC expects the Chinese automotive market to expand substantially through 2015, driving adoption of MCU solutions.

5. Suppliers will invest in services value chain

While embedded hardware margins show signs of stability in 2011, it's clear to VDC that leading embedded suppliers also recognize the value their clients place on a range of services capabilities. As a result, many leading suppliers will try to differentiate by investing in critical aspects of the services value chain, from consulting capabilities to enhanced warranty and end-of-life policies.

6. FPGA and GPU will expand into a number of market segments

The medical, industrial automation and military segments provide an attractive opportunity for FPGA (Field Programmable Gate Array) devices. From imaging equipment to diagnostic devices, there is a need for adaptable health care, factory control and military C4 solutions. The programmability, flexibility and reduced NRE (non-recurring engineering) costs associated with FPGAs will lend themselves to broader adoption in these markets.

7. Tier 2/Tier 3 OEMs and ISVs will become more important

Investment in solutions requiring embedded platforms continues to rebound; however, the market will still be driven by small- to mid-sized projects. This is related to the slow return of larger, blanket purchase orders let by Tier 1 accounts and to the user community preferences for projects with smaller footprints that fit within narrower application definitions and require short, sharply defined systems integration support. These projects are tailor-made for local, expert ISVs and ISIs, as well as Tier 2/Tier 3 OEMs.

8. The market explores HaaS (Hardware as a Service) bundles

Broad market expansion and deep application penetration of remote monitoring and control capabilities will advance across a number of market segments, foretelling a broader migration to managed services solution development and deployment models in supervisory monitoring and control applications. These embedded application clouds will require local points of presence (POPs) or on-site infrastructure and hardware rolled into service level agreements (SLAs) supporting the software and service delivery portions of contracts.

9. Cross-platform processor suppliers learn to play nice

From broader, bigger, more aggressive, public licensing agreements to M&A, the market will force suppliers of CPU, FPGA and GPU (graphics processing unit) technologies to collaborate more in 2011. VDC Research's surveys of hundreds of OEMs across a number of embedded markets reveal significant growth in OEM plans to develop solutions on hybrid platforms incorporating two or more of these technologies.

10. Competition will intensify and growth will accelerate

Even if the market does not return to pre-recession levels, growth will accelerate during 2011. VDC sees virtually every vertical market growing more than five percent, and most technology categories achieving the same five percent CAGR. However, profitability results may not be so positive. Demand for stable technologies, brutal price concessions and expanded services requirements will provide opportunities for differentiation and revenues, but not necessarily margin.

11. Android to catalyze further growth in commercial Linux market

As device manufacturers take Android into new application classes beyond mobile, the commercial Linux market will experience further growth.

12. Multi-OS systems will grow in designs

More application classes will have sophisticated UI functionality that is not supported by traditional OSs and end-users will seek out multi-OS systems.

13. **Virtualization in embedded and mobile systems will increase**

Driven by hardware bill of materials savings and reduced concerns regarding additional run-time execution latencies and costs, operating system virtualization will provide increased growth opportunities, and therefore will continue to be a significant focus for many suppliers.

14. **Symbian's loss to become MeeGo's gain**

Intel's increasing focus on embedded combined with Symbian's loss of strategic direction will drive additional gains for MeeGo as Nokia turns their attention toward the Linux-based platform.

15. **OEMs to increase focus on the use of web security test tools**

Increased interaction with the cloud and web-based content by more embedded device classes will increase OEM focus on use of web security test tools.

16. **Telecom vertical will reaccelerate spend on commercial products**

The increasing burden of mobile device data usage is driving the need for investment in wireless infrastructure and the telecom vertical market will reaccelerate spending on commercial products.

17. **Microsoft will regain relevance in the mobile phone sector**

Riding the wave of Windows Phone 7 buzz, Microsoft will re-emerge as a leading player in the mobile phone arena.

18. **Another acquisition to come?**

Following a string of high profile acquisitions in 2009/2010, VDC anticipates yet another major embedded real-time operating system supplier will get acquired in 2011.

Embedded Linux

From wiki (http://en.wikipedia.org/wiki/Embedded_Linux)

Embedded Linux is the use of Linux in embedded computer systems such as mobile phones, personal digital assistants, media players, set-top boxes, and other consumer electronics devices, networking equipment, machine control, industrial automation, navigation equipment and medical instruments. According to survey conducted by Venture Development Corporation, Linux was used by **18%** of embedded engineers.

Linux has been ported to a variety of processors not always suited for use as the processor of desktop or server computers, such as various CPUs including **ARM**, AVR32, Blackfin, ETRAX CRIS, FR-V, H8300, IP7000 M32R, m68k, MIPS, mn10300, **PowerPC**, SuperH, or Xtensa processors, as an alternative to using a proprietary operating system and toolchain.

The advantages of embedded Linux over proprietary embedded operating systems include no royalties or licensing fees, a stable kernel, a support base that is not restricted to the employees of a single software company, and the ability to modify and redistribute the source code. The disadvantages include a comparatively larger memory footprint (kernel and root file system), complexities of user mode and kernel mode memory access and complex device drivers framework.

There are non-proprietary embedded operating systems that share the open-source advantages of Linux, without the memory requirements that make Linux unsuitable for many embedded applications.

Embedded Systems related pages

1. Embedded Systems Programming I - Introduction
(/cplusplus/embeddedSystemsProgramming.php)
2. Embedded Systems Programming II - gcc ARM Toolchain and Simple Code on Ubuntu and Fedora
(/cplusplus/embeddedSystemsProgramming_gnu_toolchain_ARM_cross_compiler.php)
3. Embedded Systems Programming III - Eclipse CDT Plugin for gcc ARM Toolchain
(/cplusplus/embeddedSystemsProgramming_GNU_ARM_ToolChain_Eclipse_CDT_plugin.php)
4. Memory-mapped I/O vs Port-mapped I/O
(/Embedded/memory_mapped_io_vs_port_mapped_isolated_io.php)
5. Interrupt & Interrupt Latency
(/Embedded/hardware_interrupt_software_interrupt_latency_irq_vs_fiq.php)
6. Little Endian/Big Endian & TCP Sockets (/Embedded/Little_endian_big_endian_htons_htonl.php)
7. Bit Manipulation (/cplusplus/quiz_bit_manipulation.php)
8. Linux Processes and Signals (/Linux/linux_process_and_signals.php)
9. Linux Drivers 1 (/Linux/linux_drivers_1.php)



0 Comments bogotobogo.com Login ▾ Recommend  Share

Sort by Best ▾



Start the discussion...

Be the first to comment.

 Subscribe  Add Disqus to your site Add Disqus Add  Privacy**DISQUS**

CONTACT

BogoToBogo
contactus@bogotobogo.com (mailto:#)

FOLLOW BOGOTOBOGO

f (<https://www.facebook.com/KHongSanFrancisco>) **🐦**
(<https://twitter.com/KHongTwit>) **g⁺**
(<https://plus.google.com/u/0/+KHongSanFrancisco/posts>)

ABOUT US (/ABOUT_US.PHP)

contactus@bogotobogo.com (mailto:#)

Golden Gate Ave, San Francisco, CA 94115

Golden Gate Ave, San Francisco, CA 94115

Copyright © 2016, bogotobogo
Design: Web Master (<http://www.bogotobogo.com>)