```
 1 #include <iostream>
 2
 3 void println(size_t v) {
 4     std::cout << v << ' ';
 5 }
 6
 7 void println(bool b) {
 8     std::cout << std::boolalpha << b << ' ';
 9 }
10
11 int main() {
12     println(sizeof(bool));
13     println(sizeof(char));
14     println(sizeof(short));
15     println(sizeof(int));
16     println(sizeof(long));
17     println(sizeof(long long));
18     println(sizeof(float));
19     println(sizeof(double));
20     println(sizeof(long double));
21
22     println(sizeof(char) == 1);
23     println(sizeof(bool) < sizeof(int));
24     println(sizeof(short) < sizeof(int));
25     println(sizeof(int) == sizeof(long));
26     println(sizeof(unsigned int) == sizeof(signed int));
27     println(sizeof(long) >= 4);
28 }
```

What will this code print out?

On my machine (MacBook Pro, Intel Core Duo, OS X 10.4, gcc 4.0.1) I get:

```
g++ -Wall scratch.cpp && ./a.out
1 1 2 4 4 8 4 8 16 true true true true true true

[§4.6,p75]
1 == sizeof(char) =< sizeof(short) =< sizeof(int) <= sizeof(long)
1 <= sizeof(bool) <= sizeof(long)
sizeof(char) <= sizeof(wchar_t) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
sizeof(N) == sizeof(signed N) == sizeof(unsigend N)
char is at least 8 bits
short and int is at least 16 bits
long is at least 32 bit.

Notice that "long long" is not really in the C++ standard (yet) but
it seems like most compilers accepts it.
```

```
 1 #include <iostream>
 2
 3 void p1(int i) { std::cout << i << std::endl; }
 4
 5 template<typename T> void p2(T x) { std::cout << x << std::endl; }
 6
 7 int main() {
 8
 9     char c = 128;
10     p1(c);
11     p2(c);
12
13     unsigned int i = -1;
14     p1(i);
15     p2(i);
16
17     long l = -1;
18     p1(l);
19     p2(l);
20
21     double f = 3.14;
22     p1(f);
23     p2(f);
24
25     bool b = -1;
26     p1(b);
27     p2(b);
28 }
```

What might happen if you compile, link and run this program?

```
g++ -Wall scratch.cpp && ./a.out
-128
\200
-1
4294967295
-1
-1
3
3.14
1
1
```

[§4.3,p72] Unfortunately, which choice is made for a plain char is implementation-defined.

```
 1 #include <iostream>
 2 #include <string>
 3
 4 enum day { mon, tue, wed, thu, fri, sat, sun };
 5
 6 std::string toString(day d) {
 7     std::string s;
 8     switch(d) {
 9     case mon: return "mon";
10     case tue: return "tue";
11     case wed: return "wed";
12     case thu: return "thu";
13     case sat: return "sat";
14     case sun: return "sun";
15     }
16     return "x";
17 }
18
19 int main() {
20     std::cout << toString(mon) << std::endl;
21     std::cout << toString(wed) << std::endl;
22     std::cout << toString(day(4)) << std::endl;
23     std::cout << toString(day(8)) << std::endl;
24 }
```

What might happen if you compile, link and run this program?

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'std::string toString(day)':
scratch.cpp:8: warning: enumeration value 'fri' not handled in switch
mon
wed
x
x
```

Without -Wall, no warning is given.

What happens in line 23 is undefined [§4.8,p77]. Since mon=0 to sun=6 can be represented by 3 bits, the range is only 0:7 and thus 8 is not within the range.

How can this be done differently? Map, array, vector, ostream?

```
 1 #include <iostream>
 2
 3 int main() {
 4     char* str = "Hello World\n";
 5     char* iter, end;
 6
 7     iter = &str[0];
 8     end = &str[strlen(str)];
 9
10     while ( iter != end ) {
11         std::cout.put(*iter);
12         ++iter;
13     }
14     std::cout.flush();
15 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:8: error: invalid conversion from 'char*' to 'char'
scratch.cpp:10: error: ISO C++ forbids comparison between pointer and integer
```

Line 5 declares multiple names, and while iter is a pointer to char, end is a char and therefore you get a compilation error. Such constructs make a program less readable and shuld be avoided [§4.9.2,p80].

Line 8 is ok. [§5.3,p92] Taking a pointer to the element one beyond the end of an array is guaranteed to work.

Apart from the error in line 5, and that it is unusual to use char pointers for strings in C++, this code demonstrates a reasonable way of iterating through an array.

IMHO such declarations should be written like:

```
    char * p;
    char q;
```

Hardcore C coders prefer 'char *p', like Kernighan and Ritchie does.
Hardcore C++ coders prefer 'char* p', like Stroustrup does.
The first form seems sensible in old style C, where declaring lots of variables in the top of a block is/was common and saving visual space make sense. In C++ you tend to declare variables as close to usage as possible, and most coders never declare multiple names in a single declarations. Also, grouping the operator like Stroustrup does emphatizes the type more than the name.

However, having a space on each side of the operator (*) in declarations seems to becoming more and more popular both for C and C++ for several reasons.

```
 1 #include <iostream>
 2
 3 int x = 42;
 4
 5 int main() {
 6     int x = 43;
 7     {
 8         int x = 44;
 9         std::cout << x << std::endl;
10     }
11 }
```

What might happen if you try to compile, link and run this code?

```
g++ scratch.cpp && ./a.out
44

g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:6: warning: unused variable 'x'
44

g++ -Wshadow -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:6: warning: declaration of 'x' shadows a global declaration
scratch.cpp:3: warning: shadowed declaration is here
scratch.cpp:8: warning: declaration of 'x' shadows a previous local
scratch.cpp:6: warning: shadowed declaration is here
scratch.cpp:6: warning: unused variable 'x'
44

How can you make this code print 42? by using ::x in line 9
Can you make this code print 43? ([§4.9.4,p82] There is no way to use a hidden local name)
```

```
 1 #include <iostream>
 2
 3 int a;
 4
 5 namespace {
 6     int b;
 7 }
 8
 9 int main() {
10     static int c;
11     int d;
12     int * e = new int();
13
14     std::cout << a << std::endl;
15     std::cout << b << std::endl;
16     std::cout << c << std::endl;
17     std::cout << d << std::endl;
18     std::cout << *e << std::endl;
19 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
0
0
0
0
0
```

The first three lines should always be 0, [§4.9.5,p83] static objects (global, namespace, local static) is initialized to 0, while local variables and objects created on the free store are not initialized by default.

The variables in line 17 and 18 does not have a well-defined value. It could be 0, as in this case, or it could be something else.

Notice that odd things might happen when you add compiler directives. Eg,

```
g++ -O2 -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:11: warning: 'd' is used uninitialized in this function
0
0
0
-1599018720
0
```

Some people claim that explicit initializing static objects is confusing and should not be done. Eg, they prefer:

```
  static int x;         over            static int x = 0;
```

Other claims that static objects should also be explicitly initialized for clearity.

```
 1 #include <iostream>
 2
 3 void p(int * p) {
 4     std::string separator = "";
 5     for ( int i = 0 ; i < 4 ; ++i ) {
 6         std::cout << separator << p[i];
 7         separator = ",";
 8     }
 9     std::cout << std::endl;
10 }
11
12 int main() {
13     int a[4];
14     int b[4] = {1,2};
15     int c[4] = {};
16     static int d[4];
17     static int e[] = {1,2,3,4};
18
19     p(a);
20     p(b);
21     p(c);
22     p(d);
23     p(e);
24 }
```

```
g++ -Wall scratch.cpp && ./a.out
-1878716180,-1881117246,0,-1073743988
1,2,0,0
0,0,0,0
0,0,0,0
1,2,3,4
```

[5.2.1,p89] if the initializer supplies too few elements, 0 is assumed for the remaining array elements.

A small note on line 4 - some people will claim that initializing a string with
and empty string is nonsense. Eg, they would rather write

```
  std::string separator;
```

The idiom used in line 4 to 8 is just one out of many idioms that can be used
to print out a list of elements. (and it is not necessarily a very good idiom)

```
 1 #include <iostream>
 2
 3 int main() {
 4     char a[] = "Foo";
 5     char * b = "Bar";
 6
 7     std::cout << a << " " << b << std::endl;
 8
 9     a[0] = 'Z';
10     std::cout << a << " " << b << std::endl;
11
12     b[0] = 'C';
13     std::cout << a << " " << b << std::endl;
14 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
Foo Bar
Zoo Bar

Compilation exited abnormally with code 138 at Sun Sep  9 23:13:34
```

Note the difference between line 4 and line 5. In line 4, the string literal is copied into the array a and stored on stack. In line 5, b just points to a statically allocated const array of characters. [5.2.2,p90]

```
 1 #include <iostream>
 2
 3 int main() {
 4     char a[] = "Hello " "World";
 5
 6     for ( int i=0; a[i] != 0; ++i ) {
 7         std::cout.put(a[i]);
 8     }
 9     std::cout.put('\n');
10
11     for ( char * p = a; *p != 0; ++p ) {
12         std::cout.put(*p);
13     }
14     std::cout << std::endl;
15 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
Hello World
Hello World
```

Line 6-8 and 12-14 are saying the same thing in
two different ways.

see also §5.3.1,p92-93

```
 1 int main() {
 2     int i = 42;
 3
 4     int * const a = &i;
 5     *a = 43;
 6     a = &i;
 7
 8     const int * b = &i;
 9     *b = 44;
10     b = a;
11
12     const int * const c = &i;
13     *c = 45;
14     c = a;
15 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:6: error: assignment of read-only variable 'a'
scratch.cpp:9: error: assignment of read-only location
scratch.cpp:13: error: assignment of read-only location
scratch.cpp:14: error: assignment of read-only variable 'c'

[5.4.1,p96] Some people find it helpful to read such declarations right-to-left.

For example,
  a is a const pointer to int
  b is a pointer to an int const
  c is a const pointer to an int const
```

```
 1 #include <iostream>
 2
 3 struct X { char * a; char b[6]; int c; };
 4
 5 std::ostream & operator<<(std::ostream & os, const X & x) {
 6     return os << x.a << " " << x.b << " " << x.c;
 7 }
 8
 9 int main() {
10     X x = {"Hello", "World", 42};
11     std::cout << x << std::endl;
12     std::cout << sizeof(X) << std::endl;
13 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
Hello World 42
16
```

[5.2,p102] The notation used for initializing arrays can also be used for initializing variables of structure types.

[5.2,p102] The size of an object of a structure type is not necessarily the sum of the sizes of its members. This is because many machines require objects of certain types to be allocated on architecture-dependent boundaries or handle such objects much more efficiently if they are.

On my machine the sizeof(X) is 16. Because

```
  pointer(4), char(6), padding(2), int(4)
```

The padding is added due to architecture-dependent boundaries.

```
1 #include <iostream>
2
3 int main(int argc, const char ** argv) {
4     std::cout << argc;
5     for ( const char ** p = argv; *p != NULL; ++p ) {
6         std::cout << " " << (*p);
7     }
8     std::cout << std::endl;
9 }
```

Given that this code is compiled, linked and executed like this:

```
g++ -o foo foo.cpp
./foo bar gaz
```

What will be printed out. Please comment the code.

```
g++ -o foo foo.cpp && ./foo bar gaz
3 ./foo bar gaz

This is platform dependant. If the environment cannot pass arguments,
then argc is set to 0.

The first argument, argv[0], is the name of the program as it occurs on
the command line.

[6.1.7,p117] argv[argc] == 0

Although 'char ** argv' and 'char * argv[]' are both common and
works, it is probably better to write 'char * argv[]'.

[5.1.1,p88] In C, it has been popular to define a macro NULL to represent the null
pointer. Because of C++'s tighter type checking, the use of plain 0, rather than
any suggested NULL macro, leads to fewer problems.

A pointer not pointing to something is 0, so it should really be
written as *p != 0, and perhaps even *p, and due to idiomatic reasons
you might find the loop written as:

  for ( const char ** p = argv; *p++; ) {
    std::cout << " " << (*p);
  }
```

```
 1 #include <iostream>
 2
 3 int foo(int a) {
 4     std::cout << a;
 5     return a;
 6 }
 7
 8 void bar(int b, int c) {
 9     std::cout << b << c;
10 }
11
12 int main() {
13     int x = foo(5) + foo(3);
14     foo(x);
15
16     int y[4] = {};
17     int i=1;
18     y[i] = i++;
19     foo(y[1]);
20
21     bar(i++, i++);
22 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:18: warning: operation on 'i' may be undefined
scratch.cpp:21: warning: operation on 'i' may be undefined
538132
```

Without -Wall, no warning is given

[6.2.2,p122] The order of evaluation of subexpressions within an expression is undefined.

Line 13 might result in 35 instead of 53 [6.2.2,p122]

But line 14 will always give 8 as printout

Line 18 gives an undefined result, thus the output from line 19 is
unknown, maybe 0, maybe 1, or maybe something strange. [6.2.2,p123]

Line 21 might result int 23, just as well as 32

```
 1 #include <iostream>
 2
 3 int main() {
 4
 5     int x = 4;
 6     if ( 2 <= x <= 8 ) {
 7         std::cout << "a" << std::endl;
 8     } else {
 9         std::cout << "b" << std::endl;
10     }
11
12     if( x == 12 & 7 ) {
13         std::cout << "c" << std::endl;
14     } else {
15         std::cout << "d" << std::endl;
16     }
17
18     if( x = 4 ) {
19         std::cout << "e" << std::endl;
20     } else {
21         std::cout << "f" << std::endl;
22     }
23
24 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:18: warning: suggest parentheses around assignment used as truth value
a
d
e
```

The condition on line 6 really says:

```
  (2 <= x) <= 8
```

while line 12 says

```
  (x == 12) & 7
```

and line 18 is the classical assignment instead
of comparison.

```
 1 #include <iostream>
 2
 3 class Foo {
 4     int value;
 5 public:
 6     Foo() : value(42) { std::cout << "a"; }
 7     ~Foo() { std::cout << "b"; }
 8     Foo(const Foo & f) { std::cout << "c"; value = f.value; }
 9     Foo & operator=(const Foo & f) { std::cout << "d"; value = f.value; return *this;
10     Foo operator++(int) { std::cout << "e"; Foo old(*this); ++*this; return old;}
11     Foo & operator++() { std::cout << "f"; value += 4; return *this; }
12 };
13
14 int main() {
15     Foo f1;
16     std::cout << "-";
17     ++f1;
18     std::cout << "-";
19     f1++;
20     std::cout << "-";
21     Foo f2 = f1;
22     std::cout << "-";
23     f2 = f1;
24     std::cout << "-";
25 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
a-f-ecfb-c-d-bb

This code illustrates one of the reasons why you should prefer
preincrement instead of postincrement. Preincrement is just
a single call to the overloaded operator (line 11), while
postincrement results in a object copy, preincrement and
object destruction (line 10,8,11,7). Postincrement, as
well as postdecrement, is a very complex operation.

[6.2.5p125] For example, y=++x is equivalent to y=(x+=1),
while y=x++ is equivalent to y=(t=x,x+=1,t), where t
is a varaible of the same type as x.

There are some other things to note here:
line 8: use member initialization instead
line 10: consider using 'this->operator++()' instead
```

```
 1 #include <iostream>
 2
 3 struct Foo {
 4     Foo() { std::cout << "a"; }
 5     Foo(int i) { std::cout << i; }
 6     ~Foo() { std::cout << "c"; }
 7 };
 8
 9 int main() {
10     Foo f1[3];
11     std::cout << "-";
12     Foo f2[3] = {1,2};
13     std::cout << "-";
14     Foo * f3 = new Foo[3];
15     std::cout << "-";
16     delete f3;
17     std::cout << "-";
18 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:10: warning: unused variable 'f1'
scratch.cpp:12: warning: unused variable 'f2'
a.out(1887) malloc: ***  Deallocation of a pointer not malloced: 0x300304; This could be a double free()
aaa-12a-aaa-c-cccccc
```

```
 1 #include <iostream>
 2 #include <string>
 3 #include <algorithm>
 4
 5 size_t count(char ch, std::string const & str)
 6 {
 7     size_t n = 0;
 8     char const * p = str.c_str();
 9     while (n += *p == ch, *p++ != 0)
10         ;
11     return n;
12 }
13
14
15 int main()
16 {
17     char const c[] = "Tamatawhakatangih\0angakoauaoutamateapolaiwhenuakitanaahu";
18     std::string str(c, sizeof(c));
19
20     if (size_t n = count('w', str)) {
21         std::cout << "Character found " << n << " times" << std::endl;
22     } else {
23         std::cout << "Character not found" << std::endl;
24     }
25
26     if (size_t n = std::count(str.begin(), str.end(), 'w')) {
27         std::cout << "Character found " << n << " times" << std::endl;
28     } else {
29         std::cout << "Character not found" << std::endl;
30     }
31 }
```

What might happen if you try to compile, link and run this code?

```
g++ -Wall scratch.cpp && ./a.out
Character found 1 times
Character found 2 times
```

Note the zero character inserted into c on line 17.

Using C style char pointers in C++ is not recommended, most
of the time you can solve the same task in a more elegant
and robust way using the standard library of C++.

In C++ you are encouraged to reduce the scope of variables as
much as possible. See line 20 and 26 to see how you can
declare variables in the condition. The n is only valid
inside the if blocks, including the else block.