```
1 #include<iostream>
2
3 int main()
4 {
5     std::cout << "Hello Studygroup\n";
6 }
```

What will happen if we try to compile, link and run this program? Do you have any comments to the code?

The code will compile, link, run and print out "Hello Studygroup".

Example from ch 1, page 5.

This is actually the first piece of code in the TC++PL book. Some will claim
that the missing space after include is unusual. Others will claim that a
a missing return statement is an error, but it is perfectly legal C++ and
surprisingly common.

```
1 #include <iostream>
2
3 int main() {
4     for (int i=0; i<3; i++)
5         std::cout << i;
6     for (int j=1; j<=3; ++j)
7         std::cout << j;
8     return 0;
9 }
```

What will this code print out?

It will print out 012123

ch2p26

What is the difference between postincrement and preincrement?

foo.cpp

```
1  int main() {
2      int r = 42;
3
4      for (int i=1; i<8; ++i)
5          r += i;
6
7      return r;
8  }
```

bar.cpp

```
 1  int main() {
 2      int r = 42;
 3
 4      int i=1;
 5      while (i<8) {
 6          r += i;
 7          ++i;
 8      }
 9
10      return r;
11  }
```

Both these programs will return 70. But do you expect the resulting binary executable to be exactly the same? Eg,

```
g++ -S foo.cpp
g++ -S bar.cpp
diff foo.s bar.s
```

There is a simple mapping from a for-loop to a while-loop. Eg,

```
for ( for-init-statement condition_opt ; expression_opt ) statement
for ( int i=1; i<4; ++i ) { std::cout << i; }
```

to

```
{ while-init-statement; while (condition) statement }
{ int i=1; while (i<4) { std::cout << i; ++i; } }
```

So if we write the while-loop inside a block in bar.cpp, then
foo.cpp and bar.cpp should produce exactly the same code.

Note that the scoping rules for variables have changed from
classic C++ to modern C++. Eg,

```
int a;
for(int i=0; i<4; i++)
  ;
a = i;
```

does not compile with C++98 because the scope of i is only
inside the for-loop, just as if:

```
int a;
{
  for(int i=0; i<4; i++)
    ;
}
a = i; // error
```

and i is not declared outside the for-loop.

stack.h

```
1  namespace Stack {
2      void push(char);
3      char pop();
4
5      class Overflow {};
6      class Underflow {};
7  }
```

main.cpp

```
1  #include "stack.h"
2
3  int main() {
4      Stack::push('H');
5      Stack::push('e');
6      Stack::push('i');
7      Stack::pop();
8      Stack::pop();
9      Stack::push('e');
10     Stack::push('l');
11     Stack::push('l');
12     Stack::push('o');
13     return 0;
14 }
```

stack.cpp

```
1  #include <iostream>
2  #include "stack.h"
3
4  namespace Stack {
5      const int max_size = 4;
6      char v[max_size];
7      int top = 0;
8  }
9
10 void Stack::push(char c) {
11     if ( top == max_size )
12         throw Overflow();
13     std::cerr << "push: " << c << std::endl;
14     v[top++] = c;
15 }
16
17 char Stack::pop() {
18     if ( top == 0 )
19         throw Underflow();
20     char c = v[--top];
21     std::cerr << "pop:  " << c << std::endl;
22     return c;
23 }
```

```
g++ -Wall main.cpp stack.cpp && ./a.out
```

What will happen if you try to compile, link and run this code? Please comment this code.

```
g++ -Wall main.cpp stack.cpp && ./a.out
push: H
push: e
push: i
pop:  i
pop:  e
push: e
push: l
push: l
terminate called after throwing an instance of 'Stack::Overflow'
```

Naming conventions. Stroustrup and others seems to prefer
names starting with a capital letter for classes and namespaces.
Functions and variables starts with lower letter. .h for
header files, and .cpp for C++ files are quite common.

Notice the order of include statements in stack.cpp and
main.cpp. At least in the stack.cpp it is important that
we include "stack.h" first, in order to avoid undetected
dependencies to iostream.

Notice the similarities between namespaces and classes.
According to Bjarne (p32): Where there is no need for more
than one object of a type, the data-hiding programming
style using modules suffices.

ch2p26-29

Creating singletons this way is not necessarily a good
idea. Even if you might have just one object of a type
in your runtime system, it might be useful to be able
to create several objects during development and testing.

stack.h

```
 1 class Stack {
 2     char * v;
 3     int top;
 4     int max_size;
 5
 6 public:
 7     class Underflow {};
 8     class Overflow {};
 9     class Bad_size {};
10
11     Stack(int s);
12     ~Stack();
13
14     void push(char c);
15     char pop();
16 };
```

main.cpp

```
 1 #include "stack.h"
 2
 3 int main() {
 4     Stack s(4);
 5     s.push('H');
 6     s.push('e');
 7     s.push('i');
 8     s.pop();
 9     s.pop();
10     s.push('e');
11     s.push('l');
12     s.push('l');
13     s.push('o');
14     return 0;
15 }
```

stack.cpp

```
 1 #include "stack.h"
 2 #include <iostream>
 3
 4 Stack::Stack(int s) {
 5     top = 0;
 6     if ( s < 0 || s > 16 )
 7         throw Bad_size();
 8     max_size = s;
 9     v = new char[s];
10 }
11
12 Stack::~Stack() {
13     delete v;
14 }
15
16 void Stack::push(char c) {
17     if ( top == max_size )
18         throw Overflow();
19     std::cerr << "push: " << c << std::endl;
20     v[top++] = c;
21 }
22
23 char Stack::pop() {
24     if ( top == 0 )
25         throw Underflow();
26     char c = v[--top];
27     std::cerr << "pop:  " << c << std::endl;
28     return c;
29 }
```

What will happen if you try to compile, link and run this code?
Please comment this code.

```
g++ -Wall main.cpp stack.cpp && ./a.out
push: H
push: e
push: i
pop:  i
pop:  e
push: e
push: l
push: l
terminate called after throwing an instance of 'Stack::Overflow'

There is a lot of issues to discuss in this code. Eg,
- if you new an array, you must delete an array
- if your class needs a destructor, you probably also need
  implement or hide the copy constructor and assignment operator.
- initializing members should be done in an initializer list
- the size of the stack should probably be given as size_t

ch2p26-29
```

```
 1 #include <iostream>
 2
 3 class U {
 4     int a;
 5 };
 6
 7 class V {
 8     int a;
 9     int b;
10 };
11
12 class W {
13     int a;
14     int b;
15     void foo() { a = 3; }
16 };
17
18 class X {
19     int a;
20     int b;
21     void foo() { a = 3; }
22     void bar() { b = 4; }
23 };
24
25 class Y { };
26
27 class Z {
28     void foo() {}
29 };
30
31 int main() {
32     std::cout << sizeof(U) << std::endl;
33     std::cout << sizeof(V) << std::endl;
34     std::cout << sizeof(W) << std::endl;
35     std::cout << sizeof(X) << std::endl;
36     std::cout << sizeof(Y) << std::endl;
37     std::cout << sizeof(Z) << std::endl;
38 }
```

What will this code print out when executed?

This is of course compiler and processor dependant.
On my machine (MacBook Pro, Intel Core Duo, OS X 10.4, gcc 4.0.1) I get:

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

Why do you think that the size of an empty class is 1? To avoid
having two objects with the same address, eg:

```
  Y y1;
  Y y2;
  assert( &y1 != &y2 );
```

as described in:

  http://www.research.att.com/~bs/bs_faq2.html#sizeof-empty

```
 1 #include <iostream>
 2
 3 class U {
 4     int a;
 5     int b;
 6 };
 7
 8 class V : public U {
 9     int c;
10 };
11
12 class W : public V {
13 };
14
15 class X : public W {
16     int d;
17 };
18
19 class Y {};
20
21 class Z : public Y {
22     int e;
23 };
24
25 int main() {
26     std::cout << sizeof(U) << std::endl;
27     std::cout << sizeof(V) << std::endl;
28     std::cout << sizeof(W) << std::endl;
29     std::cout << sizeof(X) << std::endl;
30     std::cout << sizeof(Y) << std::endl;
31     std::cout << sizeof(Z) << std::endl;
32 }
```

What will this code print out when executed?

This is of course compiler and processor dependant.
On my machine (MacBook Pro, Intel Core Duo, OS X 10.4, gcc 4.0.1) I get:

```
g++ -Wall scratch.cpp && ./a.out
8
12
12
16
1
4
```

```
 1 #include <iostream>
 2
 3 class A {
 4 public:
 5     void foo() { std::cout << "A"; }
 6 };
 7
 8 class B : public A {
 9 public:
10     void foo() { std::cout << "B"; }
11 };
12
13 class C : public A {
14 public:
15     void foo() { std::cout << "C"; }
16 };
17
18 class D : public A {
19 };
20
21 void bar(A & a) {
22     a.foo();
23 }
24
25 int main() {
26     B b;
27     C c;
28     bar(b);
29     bar(c);
30 }
```

What will happen when you compile, link and execute this code?

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

How can you get this code to print out BC? Add virtual to line 5.
If foo in A is virtual, how can you get this code to print out
AA by just removing one character? rewrite to void(bar A a)

```
 1 #include <iostream>
 2
 3 class U {
 4     int a;
 5     int b;
 6 };
 7
 8 class V {
 9     int a;
10     int b;
11     void foo() { a = 2; }
12     void bar() { b = 3; }
13 };
14
15 class W {
16     int a;
17     int b;
18     virtual void foo() { a = 2; }
19     virtual void bar() { b = 3; }
20 };
21
22 class X {
23     int a;
24     int b;
25     virtual void foo() { a = 2; }
26 };
27
28 int main() {
29     std::cout << sizeof(U) << std::endl;
30     std::cout << sizeof(V) << std::endl;
31     std::cout << sizeof(W) << std::endl;
32     std::cout << sizeof(X) << std::endl;
33 }
```

What will this code print out when executed?

This is of course compiler and processor dependant.
On my machine (MacBook Pro, Intel Core Duo, OS X 10.4, gcc 4.0.1) I get:

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp:15: warning: ’class W’ has virtual functions but non-virtual destructor
scratch.cpp:22: warning: ’class X’ has virtual functions but non-virtual destructor
8
8
12
12
```

without -Wall, no warning is produced.

Ch2p36-37. "A common implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions. That is usually called ‘‘a virtual function table’’ or simply, a vtbl. Each class with virtual functions has its own vtbl identifying its virtual functions. ... Its space overhead is one pointer on each object of a class with virtual functions plus one vtbl for each such class."

```
 1 #include <iostream>
 2
 3 int main()
 4 {
 5     char ch = 'c';
 6     std::cout << '3';
 7     switch(ch) {
 8     case 'a':
 9         std::cout << 'a';
10     case 'b':
11         std::cout << 'a';
12     case 'c':
13         std::cout << 'c';
14     case 'd':
15         std::cout << 'd';
16     case 'e':
17         std::cout << 'e';
18     default:
19         std::cout << 'x';
20     }
21     std::cout << '4';
22     return 0;
23 }
```

What will this code print out?

```
g++ -Wall foo.cpp && ./a.out
3cdex4
```

```
 1 #include <iostream>
 2
 3 int main()
 4 {
 5     char ch = 'r';
 6     std::cout << '3';
 7     switch(ch) {
 8     case 'a':
 9         std::cout << 'a';
10     default:
11         std::cout << 'x';
12     case 'b':
13         std::cout << 'a';
14     case 'c':
15         std::cout << 'c';
16     }
17     std::cout << '4';
18     return 0;
19 }
```

What will happen when we try to compile, link and run this code?

```
g++ -Wall foo.cpp && ./a.out
3xac4
```

```
 1 #include <iostream>
 2
 3 int main()
 4 {
 5     char ch = 'r';
 6     std::cout << '3';
 7     switch(ch) {
 8     case 'a':
 9         std::cout << 'a';
10         break;
11     case 'b':
12         std::cout << 'a';
13         break;
14     case 'c':
15         std::cout << 'c';
16         break;
17     defualt:
18         std::cout << 'x';
19         break;
20     }
21     std::cout << '4';
22     return 0;
23 }
```

What will this code print out when executed?

```
g++ -Wall foo.cpp && ./a.out
foo.cpp: In function 'int main()':
foo.cpp:17: warning: label 'defualt' defined but not used
34
```

```
 1 #include <iostream>
 2 #include <list>
 3
 4 int main() {
 5     std::list<int> mylist;
 6     mylist.push_front(1);
 7     mylist.push_front(2);
 8     mylist.push_back(3);
 9     mylist.push_back(4);
10     for (std::list<int>::iterator i = mylist.begin();
11         i != mylist.end(); ++i) {
12         std::cout << *i;
13     }
14     return 0;
15 }
```

What will this code print out when executed?

```
g++ -Wall foo.cpp && ./a.out
2134
```

```
 1 #include <iostream>
 2 #include <vector>
 3
 4 int main() {
 5     std::vector<char> v;
 6     v.push_back('a');
 7     v.push_back('b');
 8     v.push_back('c');
 9     v.push_back('d');
10
11     for (int i=0; i<v.size(); ++i) {
12         std::cout << v[i];
13     }
14     std::cout << std::endl;
15
16     for (std::vector<char>::size_type i=0; i<v.size(); ++i) {
17         std::cout << v.at(i);
18     }
19     std::cout << std::endl;
20
21     return 0;
22 }
```

Please comment this code.

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:11: warning: comparison between signed and unsigned integer expressions
abcd
abcd

The first for-loop is bad because it uses the wrong type in the loop.
The size of a vector, or any STL container, is not int, it is the
size_type of the container, eg std::vector<char>::size_type.

The second for-loop is "bad" because it does it calls v.size()
several times, and also by using at() you get double range-checking.

TC++PLp53
```

```
 1 #include <iostream>
 2 #include <map>
 3 #include <string>
 4
 5 void print(std::pair<const std::string, int> & r) {
 6     std::cout << r.first << '/' << r.second << std::endl;
 7 }
 8
 9 int main() {
10     std::map<std::string, int> m;
11     m["A"] = 5;
12     m["E"] = 8;
13     m["B"] = 9;
14     m["C"] = 3;
15     for_each(m.begin(), m.end(), print);
16     return 0;
17 }
```

Please comment this code.

```
g++ -Wall scratch.cpp && ./a.out
A/5
B/9
C/3
E/8
```

A map keeps it element ordered so an iteration traverses the map in (increasing) order. [p62]

for_each is really defined in <algorithm> so it should really be in the include list.

Notice that print() does not change anything i r, so the argument is better written as const, eg

```
  void print(const std::pair<const std::string, int> & r) {
      std::cout << r.first << '/' << r.second << std::endl;
  }
```

```
 1 #include <iostream>
 2 #include <map>
 3 #include <string>
 4 #include <algorithm>
 5
 6 bool gt_5(const std::pair<const std::string, int> & r) {
 7     return r.second > 5;
 8 }
 9
10 int main() {
11     std::map<std::string, int> m;
12     m["A"] = 5;
13     m["E"] = 8;
14     m["B"] = 9;
15     m["C"] = 3;
16     int c = count_if(m.begin(), m.end(), gt_5);
17     std::cout << c << std::endl;
18     return 0;
19 }
```

What will the following print out?

The third argument to `count_if` is called a predicate. Can you think of other ways of writing such predicates?

```
g++ -Wall scratch.cpp && ./a.out
2
```

[p63]

gt_5() is a predicate. There are other ways you can write such predicates. You might write a functor. Eg,

```
  struct gt {
      int value_;
      gt(int value) : value_(value) {}
      bool operator()(const std::pair<const std::string, int> & r) {
          return r.second > value_;
      }
  };
```

and then use

```
  int c = count_if(m.begin(), m.end(), gt(5));
```

or you can make a template version. Eg,

```
  template <int T> bool gt(const std::pair<const std::string, int> & r) {
      return r.second > T;
  }
```

and then write:

```
  int c = count_if(m.begin(), m.end(), gt<5>);
```

Using for example Boost, you can probably also make a lambda version.