```
 1 #include <iostream>
 2
 3 static int a = 0;
 4 int b = 0;
 5
 6 void f(int n) {
 7     static int c = 0;
 8     while ( n-- ) {
 9         static int d = 0;
10         int e = 0;
11         std::cout << a++ << b++ << c++ << d++ << e++ << std::endl;
12     }
13 }
14
15 int main() {
16     f(3);
17     f(3);
18 }
```

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

§7.1.2 Static Variables, [p145]

```
g++ -Wall scratch.cpp && ./a.out
00000
11110
22220
33330
44440
55550
```

What might happen if you do not init the variables? All static variables, a,b,c,d will be set to 0, the initial value of e is unknown, but e will probably increase for each invokation. Eg you might get something like:

```
g++ -O1 -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'void f(int)':
scratch.cpp:10: warning: 'e' is used uninitialized in this function
00002
11113
22224
33332
44443
55554
```

When is a local variable initialized? (p145, A local variable is initialized when the thread of execution reaches its definition.)

When is a static variable initialized? (p145, A static variable will be initialized only the first time the thread of execution reaches its definition.)

What is the difference between line 3 and 4? The static on line 3 does not mean the same as the other static, it is a linkage directive. In modern C++ you should not use static like this, consider an unnamed namespace instead (p200).

```
 1 #include <iostream>
 2
 3 void f( int a, int & b, int * c ) {
 4     ++a;
 5     ++b;
 6     ++c;
 7 }
 8
 9 int main() {
10     int a = 3;
11     int b = 4;
12     int c = 5;
13     std::cout << a << ',' << b << ',' << c << std::endl;
14     f(a,b,c);
15     std::cout << a << ',' << b << ',' << c << std::endl;
16 }
```

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

§7.2, argument passing

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:14: error: invalid conversion from 'int' to 'int*'
scratch.cpp:14: error:   initializing argument 3 of 'void f(int, int&, int*)'
```

Suppose you fix line 14 with

```
    foo(a,b,&c);
```

then the following is printed:

```
g++ -Wall scratch.cpp && ./a.out
3,4,5
3,5,5
```

```
 1 #include <iostream>
 2
 3 void p(int i) { std::cout << i; }
 4 void p(char ch) { std::cout << ch; }
 5
 6 class A {
 7 public:
 8     A() { p(1); }
 9     A(const A & a) { p(2); }
10     void operator=(const A & a) { p(3); }
11     A(int i) { p(4); }
12     ~A() { p(5); }
13 };
14
15 void f(A & a) { p('f'); }
16 void g(const A & a) { p('g'); }
17 A h() { p('h'); return 1; }
18 const A & i() { p('i'); return 1; }
19
20 int main() {
21     A a;
22     p('-'); f(a);
23     p('-'); g(a);
24     p('-'); g(1);
25     p('-'); a = h();
26     p('-'); a = i();
27     p('-');
28 }
```

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

§7.2,§72, argument passing and value return

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'const A& i()':
scratch.cpp:18: warning: returning reference to temporary
1-f-g-4g5-h435-i453-5
```

What do you thing about line 10? How to fix it? (return reference to this)

Why can't overloading distinguish on return values? Two reasons:

- ambiguities for invocations that does not assign to something

- destroys arithmetic properties (??? find ref)

```
 1 #include <iostream>
 2 #include <string>
 3
 4 void p(double) { std::cout << "p(double)" << std::endl; }
 5 void p(float) { std::cout << "p(float)" << std::endl; }
 6 void p(std::string) { std::cout << "p(std::string)" << std::endl; }
 7 void p(const char *) { std::cout << "p(const char *)" << std::endl; }
 8 void p(char *) { std::cout << "p(char *)" << std::endl; }
 9 void p(long) { std::cout << "p(long)" << std::endl; }
10 void p(char) { std::cout << "p(char)" << std::endl; }
11
12 int main() {
13     p(1.9);
14     p('s');
15     p("hello");
16     p(1);
17 }
```

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

§7.4, overloaded function names

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:16: error: call of overloaded 'p(int)' is ambiguous
scratch.cpp:4: note: candidates are: void p(double)
scratch.cpp:5: note:                 void p(float)
scratch.cpp:6: note:                 void p(std::string) <near match>
scratch.cpp:7: note:                 void p(const char*) <near match>
scratch.cpp:8: note:                 void p(char*) <near match>
scratch.cpp:9: note:                 void p(long int)
scratch.cpp:10: note:                 void p(char)
```

How can you fix this by adding just one character? Eg, by changing line 16 to 'p(1L)' Then it will print:

```
g++ -Wall scratch.cpp && ./a.out
p(double)
p(char)
p(const char *)
p(long)
```

Keeping our new line 16. What will happen if you comment out line 7?

```
g++ -Wall scratch.cpp && ./a.out
p(double)
p(char)
p(char *)
p(long)
```

A literal string in C++ is "magic", it is really a "const char *" but it will be silently converted to a "char *" if needed. This is due to C compatibility. Apparently this is about to change with the next version of C++. (??? is this true?)

```
 1 #include <iostream>
 2
 3 #define MIN(a,b) (((a)<(b))?(a):(b))
 4
 5 int main() {
 6     int a=2;
 7     int b=3;
 8     int c=0;
 9     std::cout << c << a << b << std::endl;
10     c = MIN(++a,++b);
11     std::cout << c << a << b << std::endl;
12     c = MIN(++a,++b);
13     std::cout << c << a << b << std::endl;
14 }
```

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

§7.8, badly used macros

```
g++ scratch.cpp && ./a.out
023
444
656
```

```
 1 #include <iostream>
 2
 3 #define MAX(a,b) a > b ? a : b
 4
 5 int main() {
 6     int a=2;
 7     int b=3;
 8     int c=0;
 9     std::cout << c << a << b << std::endl;
10     c = 42 + MAX(a,b);
11     std::cout << c << a << b << std::endl;
12 }
```

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

§7.8, badly written macros

```
g++ -Wall scratch.cpp && ./a.out
023
223
```

```
 1 #include <iostream>
 2
 3 int a = 4;
 4
 5 namespace {
 6     int b = 5;
 7 };
 8
 9 int main() {
10     std::cout << a << std::endl;
11     std::cout << b << std::endl;
12 }
```

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

§8.2.5.1, Unnamed Namespaces

```
g++ -Wall scratch.cpp && ./a.out
4
5
```

```
 1 #include <iostream>
 2
 3 template<typename T> void p(T x) { std::cout << x; }
 4
 5 void f() { p(1); }
 6
 7 namespace A {
 8     void f() { p(2); }
 9 }
10
11 void g() { p(3); }
12
13 namespace {
14     void g() { p(4); }
15 };
16
17 int main() {
18     f();
19     g();
20 }
```

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

§8.2, Namespaces and scope resolution

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:20: error: call of overloaded 'g()' is ambiguous
scratch.cpp:11: note: candidates are: void g()
scratch.cpp:14: note:                 void<unnamed>::g()
```

How can you make this code print 13? eg, do ::g() on line 19, or comment out line 14

How can you make this code print 14? eg, comment out line 11 (is there a way to do scope resolution into an unnamed namespace?)

How can you make this code print 23? Use 'A::f()' on line 18, and '::g()' on line 19'

foo.hpp

```
1  namespace Foo {
2      void f();
3  }
```

foo.cpp

```
1  #include <iostream>
2  #include "foo.hpp"
3
4  namespace Foo {
5      void f() { std::cout << "f()" << std::endl; }
6  }
7
8  namespace {
9      void g() { Foo::f(); }
10 }
11
12 int main() {
13     g();
14 }
```

What might happen if you try to compile, link and run this program? Please criticize.

§8.2, Namespaces, defining declared members It will compile, link, run and print out f().

However, there are at least three tings to criticized:

- [p185] When defining a previously declared member of a namespace, it is safer to define it outside, rather than reopen, the namespace.

- it is recomended to including your own headers first, then stable libraries

- include guards are missing on foo.hpp [9.3.3,p216]

```
 1 #include <iostream>
 2
 3 template<typename T> void p(T x) { std::cout << x; }
 4
 5 struct A {};
 6 struct B : A {};
 7 struct C : B {};
 8 struct D : C {};
 9
10 int main() {
11     try {
12         p(1);
13         throw B();
14         p(2);
15     } catch(A a) {
16         p(3);
17         throw C();
18         p(4);
19     } catch(B & b) {
20         p(5);
21         throw D();
22         p(6);
23     } catch(const C & c) {
24         p(7);
25         throw;
26         p(8);
27     } catch(...) {
28         p(9);
29     }
30     p(0);
31 }
```

What might happen if you try to compile, link and run this program? Criticize the code.

§8.3, Exceptions

```
g++ scratch.cpp && ./a.out
scratch.cpp: In function 'int main()':
scratch.cpp:19: warning: exception of type 'B' will be caught
scratch.cpp:15: warning:    by earlier handler for 'A'
scratch.cpp:23: warning: exception of type 'C' will be caught
scratch.cpp:19: warning:    by earlier handler for 'B'
terminate called after throwing an instance of 'C'
13
```

Exceptions should be caught by copy as on line 15. Although it will often work it is recommended to catch by reference like on line 19 for several reasons (???find ref)

foo.hpp

```
1 int a;
2 const int b = 42;
3 int foo();
```

foo.cpp

```
1 #include "foo.hpp"
2
3 int a = 42;
4 int foo() {
5     return a;
6 }
```

bar.cpp

```
1 #include "foo.hpp"
2 #include <iostream>
3
4 int main() {
5     std::cout << foo() << " is " << b << std::endl;
6 }
```

What might happen if you try to compile, link and run this program using this command line:

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

§9.2, Internal and External Linkage

```
g++ -Wall foo.cpp bar.cpp && ./a.out
foo.cpp:3: error: redefinition of 'int a'
foo.hpp:1: error: 'int a' previously declared here
```

What happens if you comment out line 1 in foo.cpp? Link error

```
g++ -Wall foo.cpp bar.cpp && ./a.out
/usr/bin/ld: multiple definitions of symbol _a
/var/tmp//ccDvqXPk.o definition of _a in section (__DATA,__data)
/var/tmp//ccFahyOW.o definition of _a in section (__DATA,__common)
collect2: ld returned 1 exit status
```

What happens if you comment out line 3 in foo.cpp? Still link error

Why do you get linkage error? Because "int a;" is a definition

Why doesn't the linker complain about line:2 in foo.hpp? b is also a definition, but it has internal linkage

How can you fix this problem? Use 'extern int a;' to make it a declaration, and then you get "42 is 42"

foo.hpp

```
1 struct S {
2     char a;
3     int b;
4 };
5
6 S foo();
```

foo.cpp

```
 1 struct S {
 2     int a;
 3     char b;
 4 };
 5
 6 S foo() {
 7     S s;
 8     s.a = 64;
 9     s.b = 'A';
10     return s;
11 }
```

bar.cpp

```
1 #include <iostream>
2 #include "foo.hpp"
3
4 S foo();
5
6 int main() {
7     S s = foo();
8     std::cout << s.a << s.b << std::endl;
9 }
```

What might happen if you try to compile, link and run this program using this command line:

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

§9.2.3, The One-Definition Rule

```
g++ -Wall foo.cpp bar.cpp && ./a.out
@65
```

What will happen if you switch linke 2 and 3 in foo.cpp?

```
g++ -Wall foo.cpp bar.cpp && ./a.out
A64
```

What if you include "foo.hpp" in foo.cpp?

```
g++ -Wall foo.cpp bar.cpp && ./a.out
foo.cpp:3: error: redefinition of 'struct S'
foo.hpp:1: error: previous definition of 'struct S'
```

What if you include "foo.hpp" in foo.cpp, and compile only bar.cpp?

```
g++ -Wall bar.cpp && ./a.out
64A
```

```
 1 #include <iostream>
 2 #include <ctype.h>
 3
 4 struct X {
 5     char id;
 6     X(char ch) : id(ch) { std::cout << (char)toupper(id); }
 7     ~X() { std::cout << id; }
 8 };
 9
10 struct A : X { A() : X('a') {} };
11 struct B : X { B() : X('b') {} };
12 struct C : X { C() : X('c') {} };
13 struct D : X { D() : X('d') {} };
14 struct E : X { E() : X('e') {} };
15 struct F : X { F() : X('f') {} };
16
17 A a;
18 static B b;
19
20 int main() {
21     C c;
22     static D d;
23     {
24         static E e;
25         F f;
26     }
27     return 0;
28 }
```

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

§9.4, Initialization and termination

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

What if you replace line 27 with 'exit(0);'?
ABCDEFfedba

Notice that C is not destroyed. (see p218)

What if you replace line 27 with 'abort();'?

g++ -Wall scratch.cpp && ./a.out
ABCDEFf

Critiques:
- in modern C++ you should use #include <cctype> on line 2
- in modern C++ you should use static_case<char> to do casting on line 6
- what does static on line 18 mean? local linkage, in modern C++ use
  unnamed namespace instead
```