

SCOPE (WHERE)

- A **(program) block** is either a function, or a block statement (if, while, for, etc.), or a compound statement (i.e., a sequence of statements surrounded by braces).
- The **scope** of a variable is the portion of the program in which that variable is visible.
 - **Local variables** have **lexical scope**, i.e., they are visible only in the block in which they are defined (and only after their point of definition).

```
int main () {
    int j;
    j = 2;
    i = 3; // Error: i is not yet defined
    int i;
    i = 3;
    {
        int j,k;
        j = 10;
        cout << j; // 10
    }
    cout << j; // 2
    k = 7; // Error: k now known in this block
}
```

- **Global variables** have **module scope**, i.e., they are known in the whole module they are defined in.

EXTENT (WHEN)

- The **extent** (lifetime, storage class) of a variable is the period of time for which that variable exists.
 - **Global variables** have **indefinite** (static) extent. They appear at their point of definition and exist as long as the program is running.
 - * They are allocated on the **heap**.
 - Normal **local variables** have **definite** (automatic) extent. They cease to exist when the program gets out of the block in which they are defined.
 - * They are created anew when the corresponding block is entered again.
 - * They are allocated on the **stack**.
 - **Static local variables** are similar to local variables, except that they have **indefinite** extent, i.e., they keep their values between two runs of their block.

```
int count() {
    static int i = 0;
    i = i+1;
    return i;
}
cout << count() // 1
<< count() // 2
<< count() // 3
<< i; // Error: i has lexical scope
```

SCOPE AND EXTENT (CONT'D)

- Static variables are also allocated on the **heap**, but with additional visibility rules
 - i.e., they are not always visible, because they still have lexical scope.
- In principle, there is nothing a global variable cannot do but a static variable can.
 - Still, it is good practice to use local (static) variables whenever you can.

	Lexical scope	Module scope
Definite extent	local variables	does not exist in C++
Indefinite extent	static variables	global variables

BEYOND MODULE SCOPE

- We want to use a variable in all of the modules that form our program. What to do?

m.h	m1.cc	m2.cc
<pre>#include <iostream> using namespace std; int i = -1; void fun();</pre>	<pre>#include "m.h" void fun () { cout << "fun: i is " << i << "\n"; i = i+1; cout << "fun: i set to " << i << "\n"; }</pre>	<pre>#include "m.h" int main () { i = 0; cout << "main: i is " << i << "\n"; fun(); cout << "main: i is " << i << "\n"; }</pre>

BEYOND MODULE SCOPE

- We want to use a variable in all of the modules that form our program. What to do?

m.h	m1.cc	m2.cc
<pre>#include <iostream> using namespace std; int i = -1; void fun();</pre>	<pre>#include "m.h" void fun () { cout << "fun: i is " << i << "\n"; i = i+1; cout << "fun: i set to " << i << "\n"; }</pre>	<pre>#include "m.h" int main () { i = 0; cout << "main: i is " << i << "\n"; fun(); cout << "main: i is " << i << "\n"; }</pre>

- When we **link** the two object files we get:

```
m2.cc:3: multiple definition of 'i'
m1.o:m1.cc:3: first defined here
```

EXTERN VARIABLES

- Solution: Forget about the header, since it is included in both C++ files.
 - Instead, we define `i` in **one** module and we declare `i` **extern** in the other one:

m.h	m1.cc	m2.cc
<pre>#include <iostream> using namespace std; void fun();</pre>	<pre>#include "m.h" int i = -1; void fun () { cout << "fun: i is " << i << "\n"; i = i+1; cout << "fun: i set to " << i << "\n"; }</pre>	<pre>#include "m.h" extern int i; int main () { i = 0; cout << "main: i is " << i << "\n"; fun(); cout << "main: i is " << i << "\n"; }</pre>

output:

```
main: i is 0
fun: i is 0
fun: i set to 1
main: i is 1
```

GLOBAL AND EXTERN VARIABLES

- They are usually a **bad idea**.
- Sometimes, however there is no (easy) way around them.
 - Between writing functions with 25 arguments and using global/extern variables, choose the latter...
 - ...but do consider first the possibility of packing those 25 arguments in a suitable structure or something similar.
- Extern variables are unavoidable when you use libraries that were build this way.
 - Many system calls in Unix return `-1` on error; in order to find the actual cause of that error, you have to inspect the extern variable `errno`.
 - The library function `getopt` sets about four variables which you have to declare `extern` in order to access them.
- In a nutshell, use extern variables when they are provided by somebody else, but think twice before providing your own variables with the intention to make them extern someplace else.