Storage Classes, Scope and Linkage

Overview

Focus is on the structure of a C++ program with

- Multiple implementation files
- Variables that must be shared among the files

How to

- Compile separate files
- Link them to create an executable

Storage Classes, Scope and Linkage

Separate Compilation

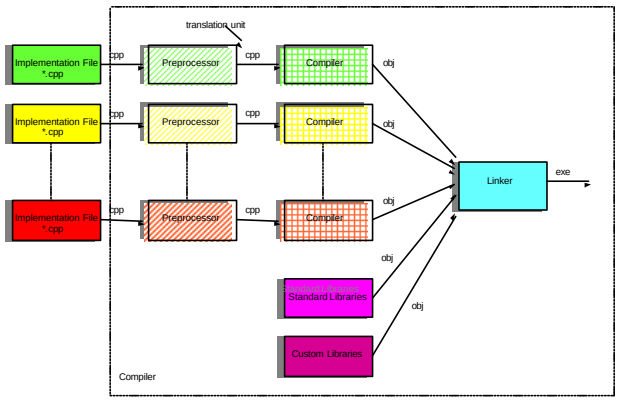
In today's large software systems many people are involved in developing same program

Each individual works on only a piece of the program

- ✓ A program comprised of all the implementation files.
- ✓ The linker combines or links compiled files into the executable program
- ✓ The entire process is called a *build*

Storage Classes, Scope and Linkage

Separate Compilation



Storage Classes, Scope and Linkage

Separate Compilation

Preprocessor locates each header file and places a copy of it in the translation unit

Replacing the *#include* of that header file

Processes any other preprocessor directives
#ifndef, *#define*, etc

Storage Classes, Scope and Linkage

Separate Compilation

After the compiler compiles the translation unit to create the object file

- The translation unit is deleted
- Each translation unit is self-contained
  - It cannot use variables or functions that are part of another translation unit

Calling *functionA()* when the code for *functionA()* is outside the translation unit produces an error...

Same is true for variables declared outside the translation unit

When this happens…the translation unit has an ***unresolved external reference***

Storage Classes, Scope and Linkage

Linking

Each implementation file is separately compiled to yield an object file by a program called the *linker*

The linker

- Reads each object file
- Copies it to the executable program

...At this time unresolved external references are resolved

Storage Classes, Scope and Linkage

Linking

When the linker fails to resolve an external reference

- It generates an *unresolved external reference* error
- Does not create the executable program

Storage Classes, Scope and Linkage

Make Files

The build process requires a place to contain the instructions for

- Which files to compile
- Lists of standard and custom libraries
- The name of the executable program
- Perhaps whether or not debugging information should be included in the executable

Such a place called a *makefile*

Storage Classes, Scope and Linkage

Make Files

The utility that …

Reads the makefile

Calls up the preprocessor, compiler, and linker

… is called the *make* utility

Storage Classes, Scope and Linkage

Standard and Custom Libraries

Compiler vendors provide libraries of compiled code to implement the C++ programming language

These are called *standard libraries*

We may write our own library to contain our favorite functions

These are called *custom libraries*

As part of the make file...

…we must specify the list of standard and custom libraries

Storage Classes, Scope and Linkage

Standard and Custom Libraries

Libraries are distributed with both a ...

- Header file
- Binary file containing the compiled code

We include

Header file in implementation file

Name of the library in the make file

Then we may make function calls into library functions.

Storage Classes, Scope and Linkage

Debug and Release Builds

A build can include or exclude information that permits a debugger to operate

- If the debugger information is excluded, the executable is much smaller, however without debugger information, we can't debug
- If the debugger information is included, the executable is much larger and slower, however debugger will operate

Storage Classes, Scope and Linkage

Debug and Release Builds

Usually a compiler switch toggles between release and debug builds

We must use caution when toggling between builds….

When we perform a debug build, we must be certain to use debug libraries in build

Conversely with a release build we must be certain to use release libraries

Reason

Memory allocators may be different between debug and release builds

Storage Classes, Scope and Linkage

Linkage, Scope, Storage Classes, and Specifiers

The terms …

- Linkage,
- Scope,
- Storage classes,
- Storage class specifiers

Often used interchangeably yet really have distinct meanings.

Storage Classes, Scope and Linkage

Linkage

There are two types of linkage *internal* and *external*

When a variable or a function has

Internal linkage

It can be used only in the implementation file in which it has been defined...

…it cannot be shared by code in another implementation file

External linkage

Means that the variable or function can be shared with another implementation file.

Storage Classes, Scope and Linkage

Scope

Scope defines visibility….

- Variables declared inside a function are only visible in that function their scope is the block of code of the function
- Variables declared outside a function - an *external variable* visible to any function in the implementation file

  These external variables are commonly called global variables

Storage Classes, Scope and Linkage

Storage Class

Storage class describes where variables are stored

C++ has three storage classes...

- ⇒ *automatic*
- ⇒ *static*
- ⇒ *freestore*

Storage Classes, Scope and Linkage

Storage Class

Automatic Storage Class

Variables with the *automatic* storage class are declared inside functions

They have internal linkage and block scope

These variables only useable in the implementation file where they are declared...

…and further only within the block of code in which they are declared

Storage Classes, Scope and Linkage

Storage Class

Static Storage Class

Variables with the *static* storage class are declared outside of any function

These are *external* variables...

- External variables are created before any use of the variable
- External variables always have external linkage
- External variables have the scope of the implementation file

Storage Classes, Scope and Linkage

Storage Class

Freestore Storage Class

Variables with the *freestore* storage class are those the programmer creates

These variables have the linkage and scope of the pointer containing the address of the variable in freestore.

These variables exist until specifically deleted

Storage Classes, Scope and Linkage

Storage Class

Storage Class Specifier

Used to provide instructions to the compiler for modifying the

Storage class, linkage, or scope of a specific variable or function

Storage class specifiers apply only to the automatic and static storage classes

Storage Classes, Scope and Linkage

Storage Class

Storage Class Specifier

Storage class specifiers are….

- auto
- register
- static
- extern

```
auto int data;
register int data;
static int data;
extern int data;
```

Storage Classes, Scope and Linkage

Storage Class

Auto Storage Class Specifier

The *auto* storage class specifier

Used only with variables to specify the *automatic* storage class

auto storage class defines

- The variable will be stored on the *stack*
- The variable will be local to the function using it
- The compiler will destroy it automatically when it is no longer needed

Storage Classes, Scope and Linkage

Storage Class

Auto Storage Class Specifier

```
auto  int  aValue;              // Error. No auto variables outside a function

void FunctionA()
{
    auto  int  a;               // Ok. auto variables go on the stack
          int  b;               // Ok. auto is assumed
}
```

Storage Classes, Scope and Linkage

Storage Class

Register Storage Class Specifier

Instructs the compiler to keep a variable in a register within the processor if possible

With the variable in a processor register not in memory

- Cannot take the address of a register variable
- Cannot have a pointer to register variable

Register storage class is a *recommendation* to the compiler

Processing may be faster

Storage Classes, Scope and Linkage

Storage Class

Register Storage Class Specifier

Time to use this storage class is when a variable is going to be accessed frequently in a very short period

Unless you are very aware of what you are doing, typically will never use register storage class

….Register variables are in a processor register

Cannot exist for the life of the program

Cannot declare a register variable outside a function

To do so requires the static storage class which would require the compiler to permanently reserve a processor register for the variable…since this is not possible, a register declaration outside a function is an error

Storage Classes, Scope and Linkage

Storage Class

Static Storage Class Specifier

The *static* storage class specifier can be used with

- Automatic or static variables
- Functions

Confusion arises because...

Name of a storage class is *static* and

Name of the storage class specifier is also *static*

Storage Classes, Scope and Linkage

Storage Class

Using the Static Storage Class Specifier

Using the static storage class specifier on a variable that normally would be automatic makes the variable static

Can use the static storage class specifier with variables declared inside functions

When the function is called the first time….

- Variable is created and initialized to zero
- Remains in existence for the remainder of the program
- Scope of the variable remains unchanged
- Can be used only in the block that declared it

Storage Classes, Scope and Linkage

Storage Class

Static Storage Class Specifier

```
void CountIt()
{
        int  count = 0;  // auto variable created on each CountIt call
        ++count;
}
```

```
void CountIt()
{
        static int  count = 0;        // variable created on first CountIt call
        ++count;
}
```


Storage Classes, Scope and Linkage

Storage Class

Static Storage Class Specifier

Static storage class specifier changes the linkage of static variables to internal linkage

Such change can *only* occur with variables declared outside functions

The scope of the variable remains unchanged

The variable can be used by any function in the implementation file

Internal linkage prevents functions in other implementation files from accessing the variable

Storage Classes, Scope and Linkage

Storage Class

Static Storage Class Specifier

Note: This use of the static storage class specifier is in C++ for backwards compatibility with C programs.

In C++, we would use a namespace to restrict access to a variable to the implementation file.

Namespaces are not covered in this course.

Storage Classes, Scope and Linkage

Storage Class

Static Storage Class Specifier

Using the static storage class specifier with a function limits the scope of the function to the implementation file containing the function

- Only other functions in the same implementation file can call it
- It is not possible to call a static function from another implementation file

```
static void functionA()
{
        // some processing
}
```

Storage Classes, Scope and Linkage

Storage Class

Extern Storage Class

The *extern* storage class specifier informs the compiler that the variable is not defined in the current implementation file

The compiler will not check to see if it is actually declared

When this implementation file is compiled

- It will have an unresolved external reference
- Reference will be left to the linker to resolve
- The location where the variable is defined is not specified

Storage Classes, Scope and Linkage

Storage Class

Extern Storage Class

Using the *extern* storage class specifier prevents the compiler from stopping build by generating an unresolved external reference error

```
void countIt()
{
    extern  int  count;          // count is declared outside this file
    ++count;
}
```

Storage Classes, Scope and Linkage

Storage Class

Extern Storage Class

The extern storage class specifier with a function works the same as with a variable

Specifies the function is defined outside the current implementation file

```
extern void countIt();          // function not defined in this file
```


Storage Classes, Scope and Linkage

Storage Class

Extern Storage Class

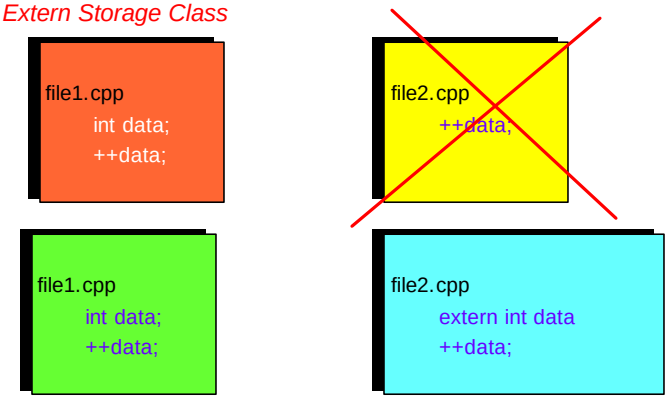
Note: Do not confuse the *extern* storage class specifier with *external* variables.

External variables are variables declared outside any function.

Storage Classes, Scope and Linkage

Storage Class

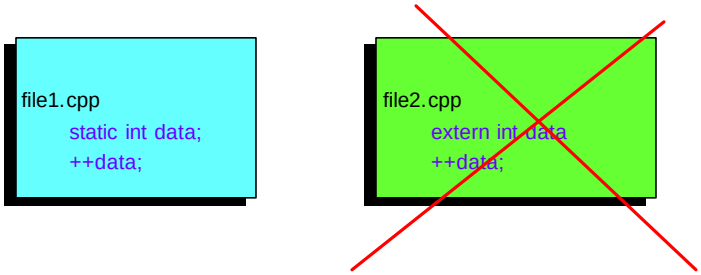
Extern Storage Class



Storage Classes, Scope and Linkage

Storage Class

Extern Storage Class



Storage Classes, Scope and Linkage

Storage Class - Summary

Specifier	Storage Class	Linkage	Scope
auto	automatic	internal	declaring block
register	automatic	internal	declaring block
--------	automatic	internal	declaring block
--------	static	external	global
static	static	internal	file or declaring block
extern	static	external	global or declaring block

Storage Classes, Scope and Linkage

Const Revisited

A global variable may be *const*:

```
const double PI  = 3.14159;
```

Because a *const* variable must be initialized when it is created….
…. PI is initialized to 3.14159 when created

If we want to share this *const* variable from another implementation file we would write

```
extern const int PI;
```

When the compiler compiles this file what value is assigned to PI?

Storage Classes, Scope and Linkage

Const Revisited

Answer is unknown ….. because PI is *extern*,

The declaration violates the *const* rule of initializing a variable with the constant value when it is created….
…as a result, the above line of code will generate an error

To use

```
const double PI  = 3.14159;
```

In each implementation file we must declare it in each implementation file
….that is *const* global variables have *internal*, or local, linkage

Storage Classes, Scope and Linkage

Functions Revisited

Where C++ Finds Functions???

When we make a function call, C++ locates the function according to this decision logic

- If the function is static
  Will use the function in the implementation file
- If the function is not static
  Will use the function from another object file
- If the function can't be found in the object file
  Library definition will be used

Storage Classes, Scope and Linkage

Functions Revisited

When user specified function prototype matches the function prototype of a library function

…The user function will be selected over the library function

Storage Classes, Scope and Linkage

Summary

In this lesson we've studied

- how to use multiple implementation files
- how to construct a header file
- how to use storage classes correctly
- how share variables among implementation files