

# Program Files and Preprocessing Directives

- Program files and header files
- A translation unit
- Linkage
- Namespaces
- Preprocessing directives
- Debugging

## Program Files

prog1.cpp

```
#include <iostream>
#include "first.h"
#include "second.h"

function definitions
global variable
definitions
etc.
```

prog2.cpp

```
#include "second.h"
#include "last.h"

function definitions
global variable
definitions
etc.
```

Adds contents of  
standard header file

## Header Files

first.h

```
class (user-type)
definitions
function declarations
etc.
```

second.h

```
class (user-type)
definitions
function declarations
etc.
```

last.h

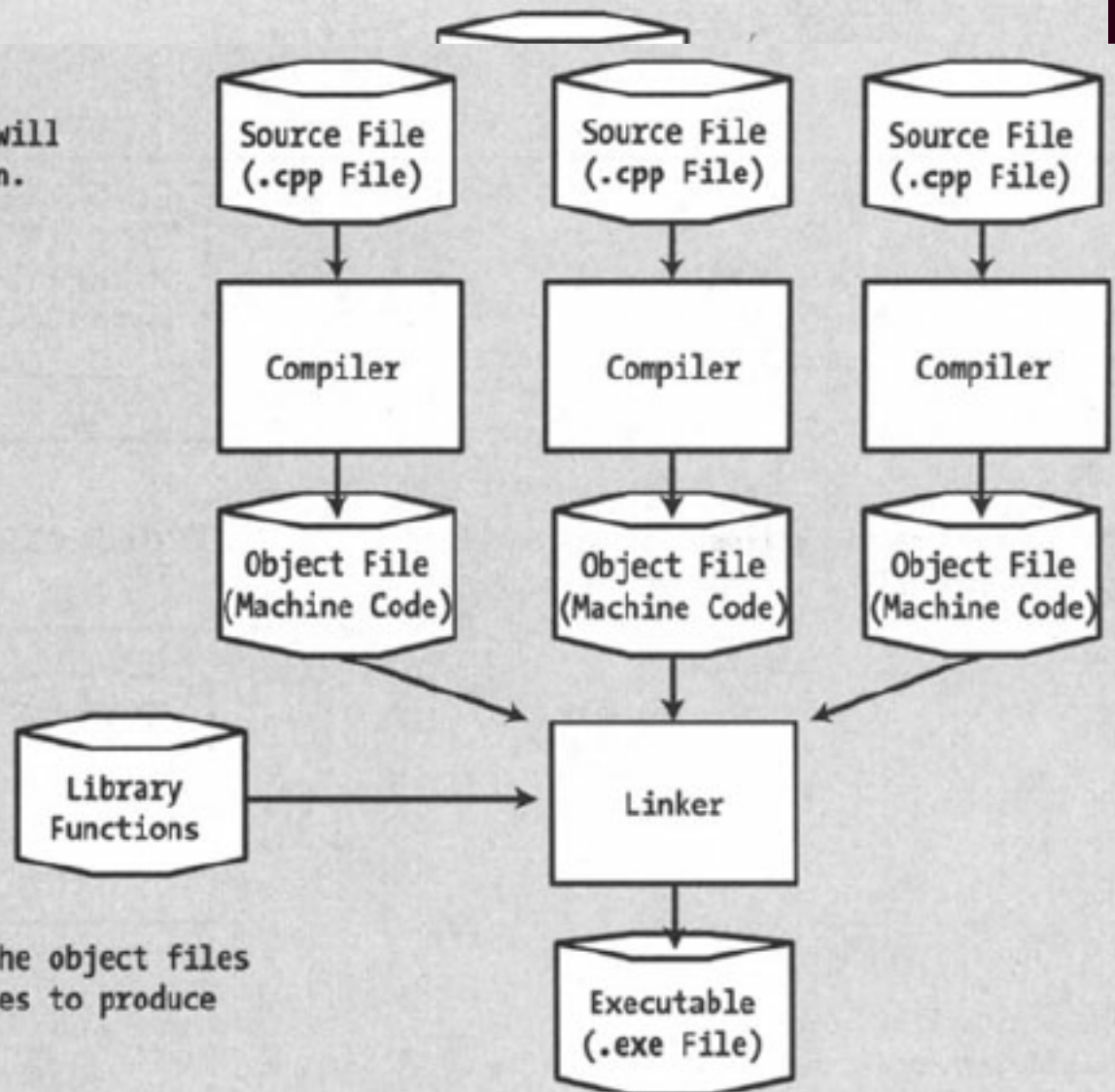
```
class (user-type)
definitions
function declarations
etc.
```

The contents of the specified  
header files are added to the  
contents of the .cpp files  
before they are compiled.

# Translation unit

The contents of header files will be included before compilation.

Each .cpp file will result in one object file.



The linker will combine all the object files plus necessary library routines to produce the executable file.

# The Scope of a Name

- Block scope (local scope)
  - Name enclosed between braces
- Name hiding

```
int main() {  
    const int limit = 10;                // Outer limit  
    std::cout << "Outer limit is " << limit << std::endl; // Outer limit  
    {  
        const int limit = 5;            // Hides limit with value 10  
        std::cout << "Inner limit is " << limit << std::endl; // Inner limit  
        for (int i = 1 ; i<= limit ; i++) // Inner limit  
            std::cout << std::endl << i << " squared is " << i*i;  
    }  
}
```

# Accessing hidden name

```
int main() {  
    const int limit = 10;                                // Outer limit  
    std::cout << "Outer limit is " << limit << std::endl; // Outer limit  
    {  
        const int limit = 5;                            // Hides limit with value 10  
        std::cout << "Inner limit is " << limit << std::endl; // Inner limit  
        for (int i = 1 ; i <= ::limit ; i++)              // Outer limit  
            std::cout << std::endl << i << " squared is " << i*i;  
    }  
}
```

# The scope of a Name

- Global scope (file scope)

```
const int limit = 10; // Global scope, also called file scope
```

```
int main() {  
    for (int i = 1 ; i<= limit ; i++)  
        std::cout << std::endl << i << " squared is " << i*i;  
}
```

```
const int limit = 10; // Global scope, also called file scope
```

```
int main() {  
    const int limit = 5; // Hides limit at global scope  
    std::cout <<"Inner limit is " << limit << std::endl; // Inner limit  
    for (int i = 1 ; i<= limit ; i++) // Inner limit  
        std::cout << std::endl << i << " squared is " << i*i;  
}
```

::limit

# Program Files and Linkage

- The ways in a translation unit are handled in the compiler/link process is determined by a property that a name can have called linkage
- Every name in a translation unit has internal linkage, external linkage, or no linkage

# Linkages

- Internal linkage
  - Access a name within the same translation unit
  - Global variables have been declared as `const`
- External linkage
  - A name can be accessed from another translation unit
  - Global variables have been declared as `non-const`
- No linkage
  - All names that are defined within a block (local names)



# External Names

```
// File2.cpp
```

```
int k = 20; // Global scope
```

```
...
```

Because of the extern declaration, this statement refers to k defined in another translation unit.

```
// File1.cpp
```

```
const int i = 10;
```

```
void function1( ) {
```

```
    const int i = 5;
```

```
    ...
```

```
    int j = ::i+5;
```

```
    ...
```

```
    int k = i+5;
```

```
}
```

```
void function2( ) {
```

```
    extern int k;
```

```
    ...
```

```
    int j = k+5;
```

```
    ...
```

```
}
```

This defines i at global scope. The use of the keyword const ensures it has internal linkage.

This refers to i at global scope because of the scope resolution operator.

This refers to the local i.

This declares that k is external to this block.

# Variable Scope

```
int main() {  
    int limit = 10;          // Illegal - redefinition!!  
    std::cout << "Local limit is " << limit << std::endl;  
    extern int limit;        // External declaration of limit  
    std::cout << "External limit is " << limit << std::endl;  
    for (int i = 1 ; i<= limit ; i++)  
        std::cout << std::endl << i << " squared is " << i*i;  
    return 0;  
}
```

```
int main() {  
    int limit = 10;          // OK - not in same block as external declaration.  
    std::cout << "Local limit is " << limit << std::endl;  
    {  
        extern int limit;    // External declaration of limit  
        std::cout << "External limit is " << limit << std::endl;  
        for (int i = 1 ; i<= limit ; i++)  
            std::cout << std::endl << i << " squared is " << i*i;  
    }  
    return 0;  
}
```

# External Linkage

- File1.cpp

```
double pi = 3.14159265;  
string days[] = {  
    "Sunday", "Monday", "Tuesday", "Wednesday",  
    "Thursday", "Friday", "Saturday"  
};
```

- File2.cpp

```
extern double pi;           // Variable is defined in another file  
extern string days[];       // Array is defined in another file
```

# Forcing Constant Variables to Have External Linkage

- Constant variables have internal linkage
- File1.cpp

```
extern const double pi = 3.14159265;  
extern const string days[] = {  
    "Sunday", "Monday", "Tuesday", "Wednesday",  
    "Thursday", "Friday", "Saturday"  
};
```

- File2.cpp

```
extern const double pi;           // Variable is defined in another file  
extern const string days[];       // Array is defined in another file
```

# Program File

- Two types of file extension .cpp, .h
- .cpp
  - Definitions
  - Main program
- .h
  - Declarations
- External variables (data10\_1.cpp, ex10\_1.cpp)

# Namespace

These variables are completely different.  
One is `calc::i` and the other is `proc::i`.

```
namespace calc {
```

```
    int i = 1;
```

```
    int max(int a, int b);
```

```
    // Plus a lot of other code
```

```
    ...
```

```
}
```

```
namespace proc {
```

```
    int i = 1;
```

```
    int max(int a, int b);
```

```
    // Plus a lot of other code
```

```
    ...
```

```
}
```

These functions are completely different.  
One is `calc::max()` and the other is `proc::max()`.

## Two Namespaces Within a Single Program

# Global Namespace

```
namespace calc
{ ... }
namespace sort
{ ... }
namespace calc
{ // extend the namespace calc }
```

- Data10\_2.cpp, ex10\_2.cpp
- Data10\_2a, ex10\_2a.cpp

# Functions in Namespace

- Using namespace and using declarations
- Using declarations
  - Compare.h, Compare.cpp, ex10\_3.cpp
  - #include "compare.h"; + using compare::max; + using compare::min; is the same work as using namespace compare;



# Template Functions and Namespaces

- Tempcomp.h, ex10\_4.cpp
- Extension namespace
  - Extent the same namespace – compare
  - Tempcomp.h, Normal.h, ex10\_5.cpp

# Namespace Aliases

- Replace the long namespace name
- Such as:
  - `namespace alias_name = original_namespace_name;`
  - `namespace SG5P3S2 =  
SystemGroup5_Process3_Subsection2;`
  - `int maxValue = SG5P3S2::max(data, size);`

# Nested Namespace

```
namespace outer {  
    double max(double* data, const int& size) { ... }  
    double min(double* data const int& size) { ... }  
    namespace inner {  
        double* normalize(double* data, const int & size) {  
            double minVal = min(data, size); // outer::min()  
        } }  
int result = outer::min(data,size);  
double* newData = outer::inner::normalize(data, size);  
// calling outer::min()  
– ex10_5a.cpp
```

# Preprocessing Directives

<b>Directive</b>	<b>Description</b>
<code>#include</code>	Supports header file inclusion
<code>#if</code>	Enables conditional compilation
<code>#else</code>	else for <code>#if</code>
<code>#elif</code>	<code>#else #if</code>
<code>#endif</code>	Marks the end of an <code>#if</code> directive
<code>#if defined (or #ifdef)</code>	Does something if a symbol is defined
<code>#if !defined (or #ifndef)</code>	Does something if a symbol is not defined
<code>#define</code>	Defines a symbol
<code>#undef</code>	Deletes a symbol
<code>#line</code>	Redefines the current line number and/or filename
<code>#error</code>	Outputs a compile-time error message and stop the compilation
<code>#pragma</code>	Offers machine-specific features while retaining overall C++ compatibility

# #include

- #include <standard\_library\_file\_name>  
#include <iostream>
- #include "self\_defined\_file\_name"  
#include "myheader.h"  
#include "..\header\myheader1.h"  
#include "..\user\header\myheader2.h"

# Substitutions

- Disadvantages
  - Without data type detection
  - Not belong to namespace, not in any scope
- #define identifier sequence\_of\_characters
  - #define PI 3.14159265
    - // string "3.14159265" replace all the PI tokens in code while compiler
  - #define BLACK WHITE // replace all the BLACK to WHITE tokens while compiler
  - #define BLACK //remove the identifier BLACK token

# #undef

```
#define PI 3.1412
```

```
...
```

```
#undef PI
```

```
...
```

- Code between `#define` and `#undef`, `PI` will be replaced to 3.142. However, the rest of the code remain `PI` without replacing

# Macro Substitutions

- `#define identifier(list_of_identifiers) substitution_string`
  - E.g., `#define Print(myVar) cout << (myVar) << endl`  
`Print(ival);`  $\Rightarrow$  `cout << (ival) << endl;`
  - E.g. `#define Print(myVar, digits) cout << setw(digits) << (myVar) << endl`  
`Print(ival, 15);`  $\Rightarrow$  `cout << setw(15) << (ival) << endl;`
- Using template function or inline function

```
template<class T> inline void Print(const T& myVar, const int&
digits)
{ cout << setw(digits) << myVar << endl;    }
Print(ival, 15);
```



# Errors Cause by Macro

```
#define max(x,y) x>y ? x:y
```

```
result = max(myval, 99);
```

```
result = myval>99 ? myval:99;
```

```
result = max(myval++, 99);
```

```
result = myval++>99 ? myval++:99;
```

```
// if myval>99 then myval will increase twice
```

```
result = max((myval++), 99);
```

```
result = (myval++)>99 ? (myval++):99;
```

# Errors Cause by Macro

```
#define product(m,n) m*n
```

```
result = product(x, y+1);
```

```
// Which means result = x*y+1
```

```
#define product(m,n) ((m)*(n))
```

- Use inline function instead of macro to prevent such errors

```
Inline int product(int m, int n) { return m*n; }
```

# #if and #endif

- #if defined identifier

```
double average = 0.0;
#if defined CALCAVERAGE
int count = sizeof data/sizeof data[0];
for (int i=0; i<count; i++)
    average += data[i];
average /= count;
#endif
```

# #ifndef

- Preventing codes duplication
- #if !defined identifier
- #ifndef identifier

# #if, #else, #endif

- #if constant\_expression  
  #if CPU == Pentium4  
    cout << "Pentium4 code version." << endl;  
  // Pentium4 oriented code  
  #else  
    cout << "Older Pentium code version." <<  
    endl;  
  // Code for older Pentium processors  
  #endif

# #elif

- #elif constant\_expression  
#if LANGUAGE == ENGLISH  
#define Greeting "Good Morning."  
#elif LANGUAGE == GERMAN  
#define Greeting "Guten Tag."  
#elif LANGUAGE == FRENCH  
#define Greeting "Bonjour."  
#else define Greeting "Hi."  
#endif  
cout << Greeting << endl;

# Standard Preprocessing Macros

Macro	Description
<code>__LINE__</code>	The line number of the current source line as a decimal integer.
<code>__FILE__</code>	The name of the source file as a character string literal.
<code>__DATE__</code>	The date when the source file was processed as a character string literal in the form <code>mmm dd yyyy</code> . Here, <code>mmm</code> is the month in characters, (Jan, Feb, etc.); <code>dd</code> is the day in the form of a pair of digits 01 to 31, where single digit days are preceded by a blank; and <code>yyyy</code> is the year as four digits (such as 1994).
<code>__TIME__</code>	The time at which the source file was compiled, as a character string literal in the form <code>hh:mm:ss</code> , which is a string containing the pairs of digits for hours, minutes, and seconds separated by colons.
<code>__STDC__</code>	This is implementation dependent. It is usually defined if a compiler option has been set to compile standard C code; otherwise it is undefined.
<code>__cplusplus</code>	This will be defined to have at least the value 199711L when a C++ program is being compiled.

# Output compilation time

```
std::cout << std::endl  
          << "Program last compiled at " << __TIME__  
          << " on " << __DATE__  
          << std::endl;
```



# #error

- Diagnostic messages in preprocessing phase

```
#ifndef _cplusplus
```

```
#error "Error – Should be C++"
```

```
#endif
```

# Integrated Debuggers

- Tracing program flow
  - Stepping through
- Setting breakpoints
- Setting watches
- Inspecting program elements

# Preprocessing Directives in Debugging

- Functions.h, Functions.cpp, ex10\_6.cpp

# Assert Macro

- Diagnostic logical expressions in program
- `ex10_7.cpp`