EEC-101
Programming with C++

Module-1.7 Pre-processor Directives

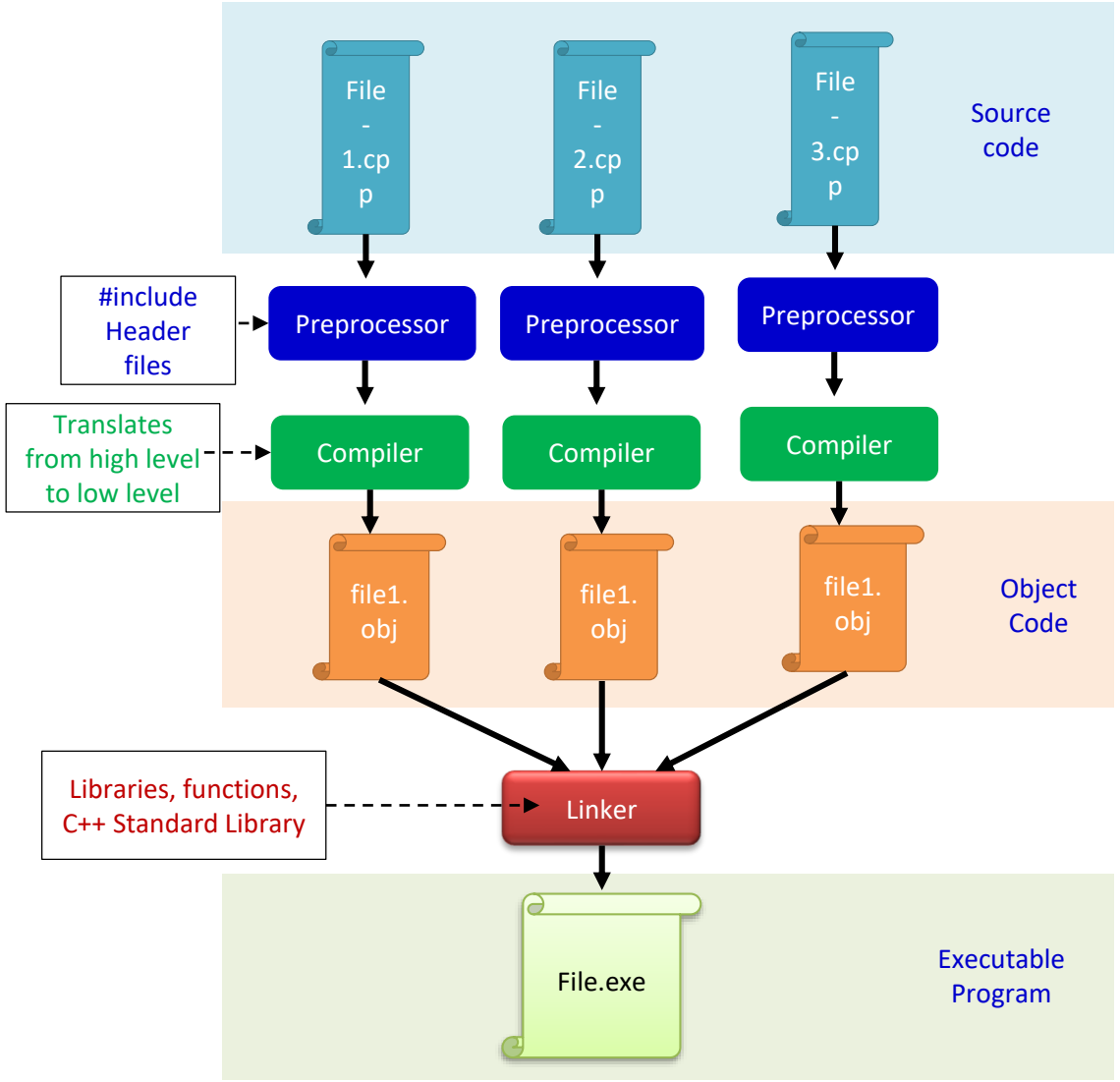
Ramanuja Panigrahi

Topics

- Concepts of algorithm & flow charts;
- Input/output,
- constants, variables
- operators;
- Naming conventions and styles;
- Conditions and selection statements;
- Looping and control structures (while, for, do-while, break and continue);
- File I/O, Header files, String processing;
- Pre-processor directives such as #include, #define, #ifdef, #ifndef;
- Compiling and linking.

Preprocessors

- Preprocessors are programs that process the source code before compilation.
- The strange name stems from the fact that compilers are sometimes called language processors, so it is natural to call such programs preprocessors.
- The source code written by programmers is first stored in a file, let the name be “file1.cpp“.
- This file is then processed by preprocessors and an expanded source code file is generated named “file1.i”. This expanded file is compiled by the compiler and an object code file is generated named “file1.obj”.
- Finally, the linker links this object code file to the object code of the library functions to generate the executable file “file1.exe”.



Preprocessor

- Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling.
- All of these preprocessor directives begin with a ‘#’ (hash) symbol.
- The ‘#’ symbol indicates that whatever statement starts with a ‘#’ will go to the preprocessor program to get executed.
- We can place these preprocessor directives anywhere in our program.
- Preprocessor directives

  #include

  #define

  #ifdef #else #endif

  #ifndef #else #endif

Why Preprocessor(s) ?

- Inclusion of other files in the file being compiled
- Definition of symbolic constant and macros
- Conditional compilation of program code
- Conditional execution of preprocessor directives

Preprocessor directives

- Preprocessor directives are not program statements but directives for the preprocessor. So do not end with ; (No semicolon)

- The preprocessor is executed before the actual compilation of code begins.

- Preprocessor directive may appear anywhere within a program
- The directive will apply only to the part of the program following its appearance

- In other words, preprocessor directives are processed fully before compilation begins. (means, the preprocessor processes all these directives before any code is generated by the statements)

Types of Preprocessors

- **There are 4 Main Types of Preprocessor Directives:**
  - Macros
  - File Inclusion
  - Conditional Compilation
  - Other directives

1. Macros

- Macros are pieces of code in a program that is given some name.
- Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code.
- The **'#define'** directive is used to define a macro or a defined constant

  – Syntax

  - #define token replacement text

  //where after preprocessing, the *token* will be expanded to its *value* in the program.

  - #define PI 3.141

#define PI 3.141

- Replaces all subsequent occurrences (except those inside a string) of the symbolic constant PI with numeric constant 3.141
- Symbolic constant enables the programmer to create a name for a constant and use the name throughout the program
- If the constant is required to be modified throughout the program, it can be modified once in #define preprocessor directive

Advantage and Disadvantage

- **Declared constants (using const keyword)** in C++ are preferred over defined constants

  const float PI= 3.14; // preferred

  # define PI 3.24 //not preferred

- Declared constants have a specified data type and are visible by name to a debugger
- Once a symbolic/defined constant is replaced with its replacement text, the replacement text is visible to a debugger
- Declared constants may require a memory location of their data type size
- Symbolic constants do not require any additional memory

Macros

- operations defined as symbols
- A macro is an operation defined in a # define preprocessor directive
- #define macro-identifier replacement-text
  - Example:
  - #define CIRCLE_AREA(x) (PI*(x)*(x)) //use parentheses

In program

area = CIRCLE_AREA(4)

is replaced by

area = (3.141*(4)*(4)) //PI comes from previous symbolic constant

Example

```
#include <iostream>
// Macro definition
#define AREA(l, w) (l * w)
using namespace std;
// Driver Code
int main()
{
    cout<< "Enter the length and width";
    float l,w,area;
    cin>>l>>w;

    // Find the area using macros
    area = AREA(l, w); // area = (l*w)

    cout<<"Area= "<<area;

    return 0;
}
```

Whenever the compiler finds AREA(l, w) in the program it replaces it with the macros definition i.e., (l*w). The values passed to the macro template AREA(l, w) will also be replaced by the statement (l*w). Therefore, AREA(10.5, 13.9) will be equal to 10.5*13.9.

```
Enter the length and width
10.5
13.9
Area= 145.95
Process returned 0 (0x0)   execution time : 9.337 s
Press any key to continue.
```

```
#define square(x) (x*x)
using namespace std;

int main()
{
    int a,b=5;
    a= square(b+5);
    cout<<a;
    return 0;
}
```

$a = n * n$

$= b+5 * b+5$

$= b + 5b + 5$

$= 6b + 5$

→ area = Area(l-2, w/3);

area = l - 2 x w/3 ;

= l - ((2 x w)/3))

#undef

- Symbolic constants and macros can be discarded using #undef preprocessor directive
- #undef undefines a symbolic or macro
- Scope of a symbolic constant or macro is from its definition until it is undefined with #undef
- Once undefined it can be, a name can be redefined with #define

Example

```
#include <iostream>
#define PI 3.142
using namespace std;
int main()
{
 //   float const PI=3.14;
  float radius =10;
  float area=PI*radius*radius;
  cout<<"area="<<area<<endl;
  #undef PI
 #define PI 3.14
 //float const PI=3.14;
 area=PI*radius*radius;
  cout<<"area="<<area;
    return 0;
}
```

// 3.142*10*10

```
area=314.2
area=314
Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

2. File inclusion

- This type of preprocessor directive tells the compiler to include a file in the source code program.
- The #include preprocessor directive is used to include the header files in the C++ program.
- There are two types of files that can be included by the user in the program:
  - Standard Header Files
    - iostream stands for standard input-output stream.
    - This header file contains definitions of objects like cin, cout, cerr, etc.
    - functions that perform string operations are in the ‘string’ file.
- Syntax: #include <file_name>
- where file_name is the name of the header file to be included. The ‘<‘ and ‘>’ brackets tell the compiler to look for the file in the standard directory.

User-defined Header Files

- When a program becomes very large, it is a good practice to divide it into smaller files and include them whenever needed. These types of files are user-defined header files.

- Syntax: #include "filename"

- The double quotes (” ”) tell the compiler to search for the header file in the source file’s directory.

3. Conditional Compilation

- Conditional Compilation in C++ directives is a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions.
- There are the following preprocessor directives that are used to insert conditional code:
  - #if Directive
  - #ifdef Directive
  - #ifndef Directive
  - #else Directive
  - #elif Directive
  - #endif Directive

- #endif directive is used to close off the #if, #ifdef, and #ifndef opening directives which means the preprocessing of these directives is completed.

Conditional inclusions

- Conditional compilation enables the programmer to control the execution of preprocessor directives and the compilation of program code
- Each conditional preprocessor directives evaluates a constant integer expression, which determine if the code will be compiled
- Cast expression and size of expression and enumeration constants can not be evaluated in preprocessor directives

- Achieved using #ifdef, #ifndef, #if, #endif, #else and #elif

- #if construct ends with #endif

- A multiple-part conditional processor construct is tested using the #elif and #else directives

Syntax

```
#ifdef macro_name
    // Code to be executed if macro_name is defined
#ifndef macro_name
   // Code to be executed if macro_name is not defined
#if constant_expr
    // Code to be executed if constant_expression is true
#elif another_constant_expr
    // Code to be excuted if another_constant_expression
is true
#else
    // Code to be excuted if none of the above conditions
are true
#endif
```

Example

```
#include <iostream>
using namespace std;
//#define DEBUG
#define PI 3.14

int main()
{
    // to check if DEBUG is defined
#ifdef DEBUG
    cout << "Debug mode is ON" << endl;
#else
    cout << "Debug mode is OFF" << endl;
#endif

    // to check if PI is defined
#ifndef PI
    cout << "PI is not defined" << endl;
#else
    cout << "PI is defined" << endl;
#endif

    return 0;
}
```

and ## Operators

Stringification Operator (#)

- The **stringizing operator (#)** is a preprocessor operator that causes the corresponding actual argument to be enclosed in **double quotation marks.**
- The # operator turns the argument it precedes into a **quoted string**. It is also known as the stringification operator.
- # operator causes a replacement text token to be converted to a string surrounded by quotes

  #define Hello(x) cout<<"Hello, "#x<<endl;

  Hello(Batch P) -> Gets expanded to cout<<"Hello, " "Batch p"<<endl;

String "Batch P" replaces #x in the replacement text

Example

```
#include <iostream>
#define Hello(x) cout<<"Hello, " #x << endl
using namespace std;
int main()
{

    Hello(EEC-101);
    Hello (54321);
    float r=5;
    Hello(r);

    return 0;
}
```

cout << "Hello, " "EEC 101" <<

"r"

```
Hello, EEC-101
Hello, 54321
Hello, r
```

Token-pasting operator (##)

- The **Token-pasting operator (##)** allows tokens used as actual arguments to be concatenated to form other tokens. It is often useful to merge two tokens into one while expanding macros.
- This is called token pasting or token concatenation.

- When a macro is expanded, the two tokens on either side of each '##' operator are combined into a single token.

The operator ## concatenates two arguments leaving no blank spaces between them:

```
#define  glue(a,b)   a ## b
glue(c,out) << "test";
```

This would also be translated into:

```
cout << "test";
```

Example

```
#include <iostream>
using namespace std;
#define sum(a,b) ((a)+(b))
#define create_var(name) name##1, name##2, name##3
#define print(name) cout<< name##1<<" "<< name##2<<" "<<name##3
int main()
{
int create_var(num); // replaced by int num1, num2, num3;
num1=22;
num2=33;
num3=sum(num1,num2);        // replaced by num3= num1+num2;
print (num);  // replaced by cout<<num1<<" "<< num2<<" "<<num
return 0;
}
```

int num1, num2, num3

```
22 33 55
Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

Predefined Symbolic Constants

- The identifiers for each predefined symbolic constant begin and end with two underscores
- __LINE__ line no. of current source code line
- __FILE__ name of source file
- __DATE__ date of source file compiled
- __TIME__ time of source file complied

__LINE__	Integer value representing the current line in the source code file being compiled.
__FILE__	A string literal containing the presumed name of the source file being compiled.
__DATE__	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.
__TIME__	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.

Example

```
// Predefined symbolic constants
#include <iostream>
using namespace std;
int main()
{
cout << "This is line number " << __LINE__;
cout << " of the file " << __FILE__ << endl;
cout << "Its compilation began " << __DATE__;
cout << " at " << __TIME__ << ".\n";
system("pause");
return 0;
}
```

```
This is the line number 6 of file C:\Users\Admin\
ocessor\pre_defined_symb_const.cpp
Its compilation began Sep 19 2024 at 12:20:43.
Press any key to continue . . .
```