C++ Software Engineering

for engineers of other disciplines

Module 4

"C++ Embedded"

2nd Lecture:Embedded Concepts

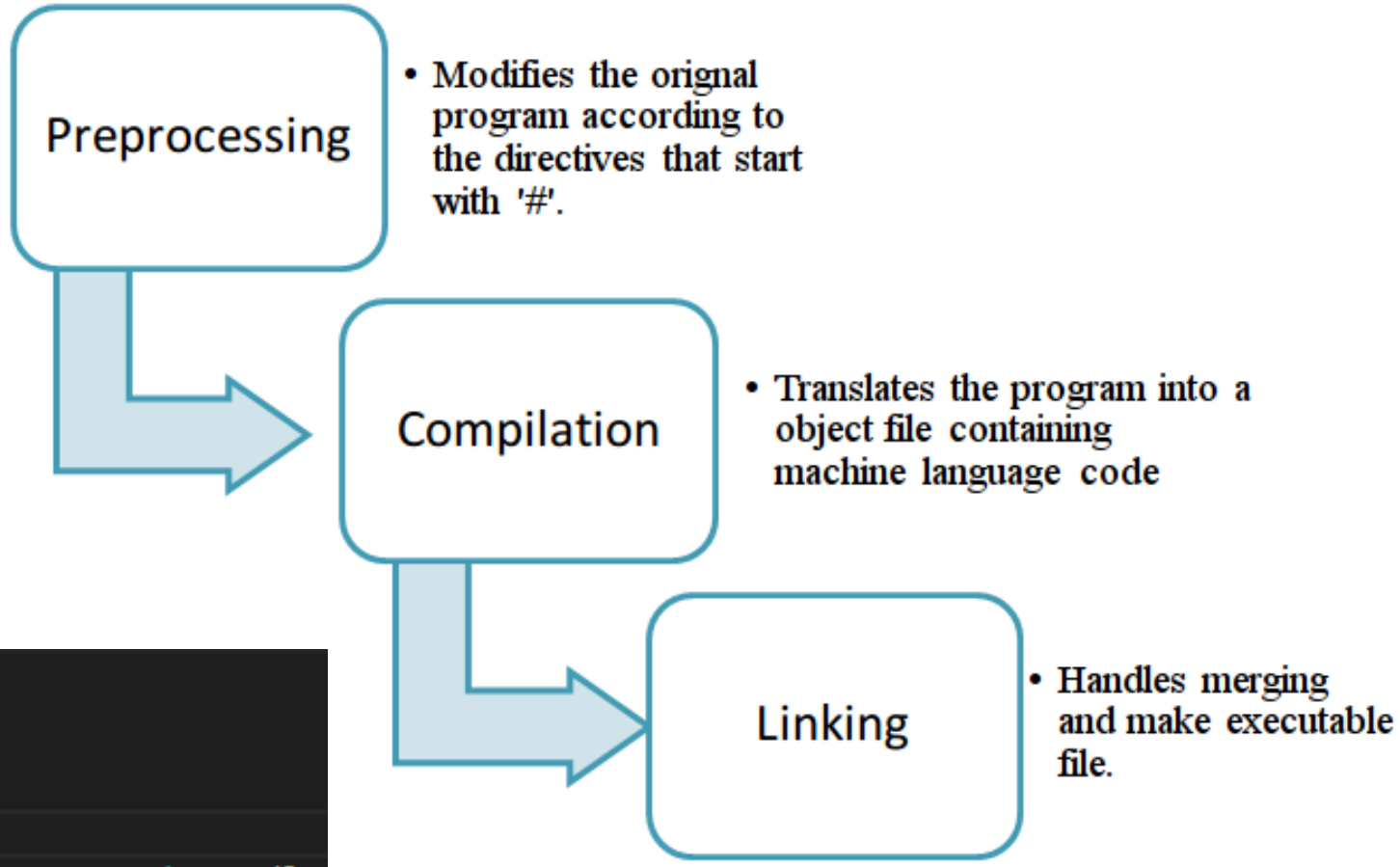
ALTEN

Spring 2022

Gothenburg, Sweden

Turnning Text To Binary – In Theory

- Software Building Procedure



http://www.cplusplus.com/articles/2v07M4Gy/Selection_101.png

helloworld.cpp > main()

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
}
```

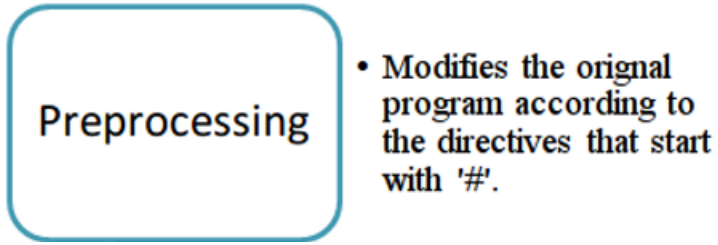
C++ Preprocessing

- Procedure prior to compilation performed by preprocessor
  - Gives the opportunity to impact compilation
  - Preprocessor prepares source code for compiler

- Preprocessor
  - Operators: **#**, #@, ##, //, /**/

  - Directives
    - They start with **#**: #define, #error, #import, #undef, #elif, #if, **#include**, #using, #else, #ifdef, #line, #endif, #ifndef, #pragma

"The preprocessor is a text processor that manipulates the text of a source file [...] the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling". https://docs.microsoft.com/en-us/cpp/preprocessor/preprocessor?view=vs-2019

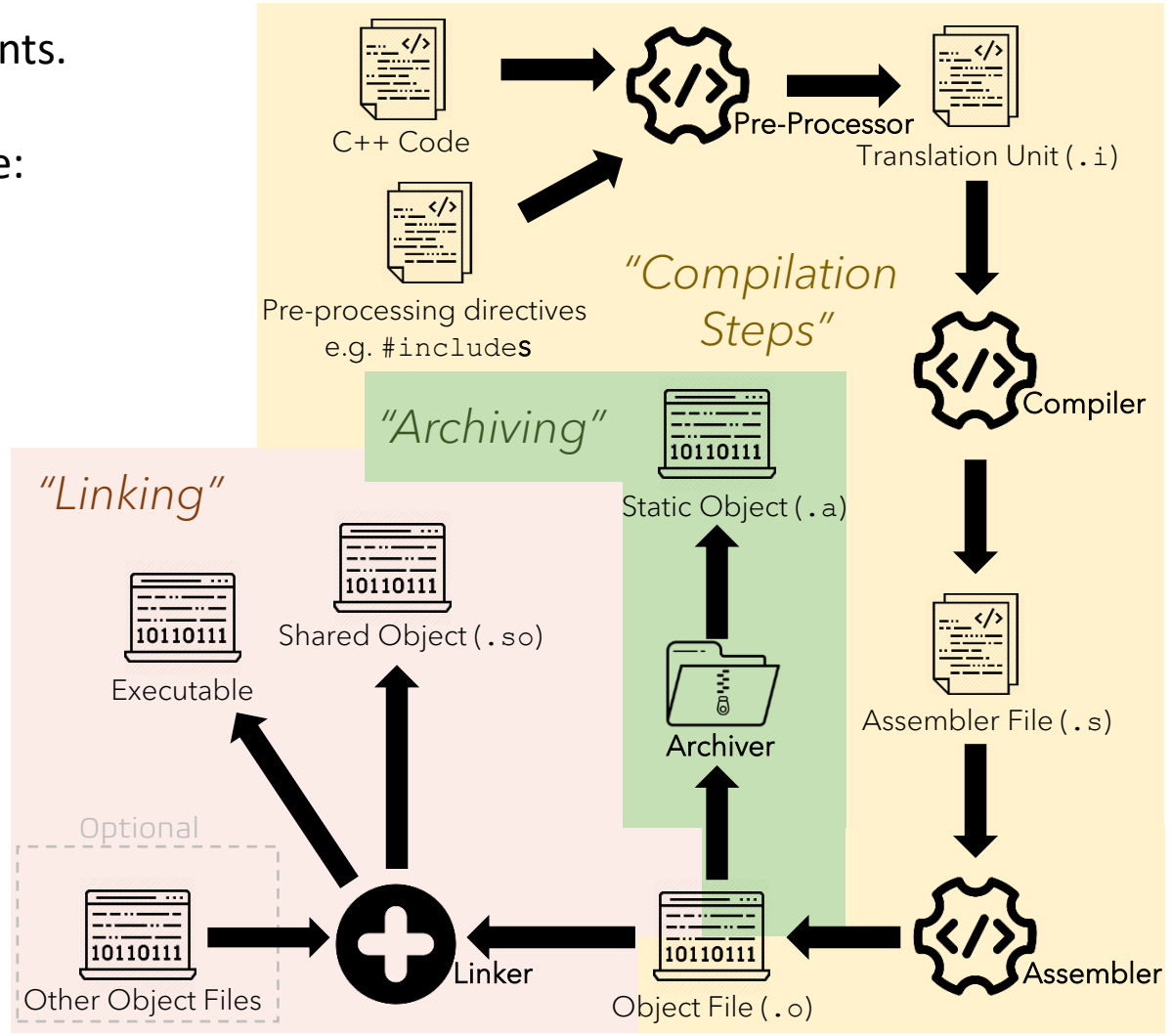



```
helloworld.cpp > main()
1   #include <iostream>
2
3   int main() {
4       std::cout << "Hello World!" << std::endl;
5   }
```

- **#include** directives copies the whole contents of the file it is provided at the exact point the directive is used. The operation could be recursive for nested includes i.e. when a file which is included has its own **#include** directive. It is common practice for better readability and maintainability to always used this directive at the very beginning of the files.

Preprocessor Directives

- Lines which start with # and are not program statements.
- They are *directives* for preprocessor, some of them are:
  - Macro definition:
    - Object-*like*
    - Function-*like*
  - Conditional inclusion



- Excessive use of Macros is considered a bad practice in C++ and other alternatives (if exist) should be favored. Code written in macros is hard (read impossible) to debug.

Preprocessor Directives

- Preprocessor directives are lines included in the code of programs preceded by a hash sign (#). These lines are not program statements but directives for the preprocessor. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.

- These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

- *1.**#include** is a way of including a standard or user-defined file in the program and is mostly written at the beginning of any C/C++ program. This directive is read by the preprocessor and orders it to insert the content of a user-defined or system header file into the following program.*
- **Syntax:**
- *#include "user-defined_file“*
- *#include <header_file>*

Source :https://www.cplusplus.com/doc/tutorial/preprocessor/

Preprocessor Directives

- *2.**#macro definitions (#define, #undef):** To define preprocessor macros we can use #define.*
- **Syntax:**
- *#define identifier replacement*
- *When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++ proper, it simply replaces any occurrence of identifier by replacement.*

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
int table2[TABLE_SIZE];
```

- *After the preprocessor has replaced TABLE_SIZE, the code becomes equivalent to:*

```
int table1[100];
int table2[100];
```

- *#define can work also with parameters to define function macros:*
- *Defined macros are not affected by block structure.*
- *A macro lasts until it is undefined with the #undef preprocessor directive:*
- *Check the difference between Macros and Functions , Inline Functions & Assert Macro*
- *.*

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

```
#define getmax(a,b) ((a)>(b)?(a):(b))

int main()
{
  int x=5, y;
  y= getmax(x,2);
  cout << y << endl;
  cout << getmax(7,x) << endl;
  return 0;
}
```

Source :https://www.cplusplus.com/doc/tutorial/preprocessor/

Preprocessor Directives

- *3. **Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)***
- *These directives allow to include or discard part of the code of a program if a certain condition is met.*
- ***#ifdef** allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:*

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

- *#ifndef serves for the exact opposite: the code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined. For example:*

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

Source :https://www.cplusplus.com/doc/tutorial/preprocessor/

Preprocessor Directives

- *3. **Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)***
- *The #if, #else and #elif (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows #if or #elif can only evaluate constant expressions, including macro expressions. For example:*

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

Preprocessor Directives

4. #Pragma

This directive is a special purpose directive and is used to turn on or off some features. These types of directives are compiler-specific i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below:

#pragma startup and #pragma exit: These directives helps us to specify the functions that are needed to run before program startup(before the control passes to main()) and just before program exit (just before the control returns from main()).**Note**: Below program will not work with GCC compilers.
Look at the below program:

```c
#include<stdio.h>

void func1();
void func2();

#pragma startup func1
#pragma exit func2

void func1()
{
    printf("Inside func1()\n");
}

void func2()
{
    printf("Inside func2()\n");
}

int main()
{
    printf("Inside main()\n");

    return 0;
}
```

Output:

```
Inside func1()
Inside main()
Inside func2()
```

The above code will produce the output as given below when run on GCC compilers:

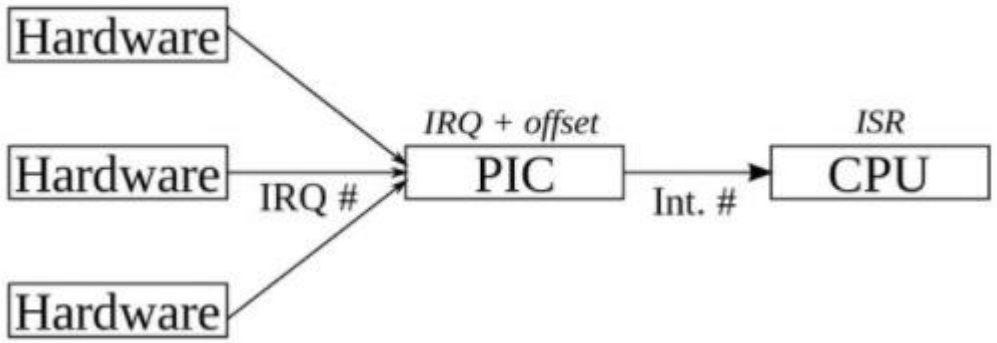
```
Inside main()
```

This happens because GCC does not support **#pragma startup or exit**. However, you can use the below code for a similar output on GCC compilers.

Source : https://https://www.geeksforgeeks.org/

Polling vs Interrupt

- *An interrupt is an event that indicates the CPU to take immediate action. There can be an interrupt to indicate the time out of a timer. Additionally, an interrupt can notify the received data packets of a networking device. When an interrupt occurs, the CPU pauses the task it is currently executing and executes the corresponding interrupt handler, which is known as Interrupt Service Routine (ISR). After handling the interrupt, it switches back to the usual tasks it was executing.*



- There are mainly two types of interrupts as hardware and software interrupt.

Source : https://pediaa.com/

Polling vs Interrupt

- Polling is the mechanism that indicates the CPU that a device requires its attention. It is a continuous act to figure out whether the device is working properly. As it is mostly used with input/output (I/O), it is also called polled I/O or software driven I/O. Polling also helps to check a device continuously for readiness. For example, take a printer. If the device is busy, the CPU performs some other task. Usually, polling often involves low-level hardware.

- However, there are some disadvantages to polling as well. Mainly, polling causes the wastage of many CPU cycles. Especially, if there are many devices to check, then the time taken to poll them could exceed the time available to service the I/O device.

Source : https://pediaa.com/

Reentrant vs Non-Reentrant Function

ALTEN

- A function is said to be reentrant if there is a provision to interrupt the function in the course of execution, service the interrupt service routine and then resume the earlier going on function, without hampering its earlier course of action. Reentrant functions are used in applications like hardware interrupt handling, recursion, etc.

- The function has to satisfy certain conditions to be called as reentrant:

1. It may not use global and static data. Though there are no restrictions, but it is generally not advised. because the interrupt may change certain global values and resuming the course of action of the reentrant function with the new data may give undesired results.
2. Should not call another **non-reentrant function**.
3. Ensure mutual exclusion of shared hardware and software resources.

```
// Both fun1() and fun2() are reentrant
int fun1(int i)
{
    return i * 5;
}

int fun2(int i)
{
return fun1(i) * 5;
}
```

```
// A non-reentrant example
// [The function depends on global variable i]

int i;

// Both fun1() and fun2() are not reentrant

// fun1() is NOT reentrant because it uses global variable i
int fun1()
{
    return i * 5;
}

// fun2() is NOT reentrant because it calls a non-reentrant
// function
int fun2()
{
return fun1() * 5;
}
```

Source : https://https://www.geeksforgeeks.org/

Synchronous vs Asynchronous Function

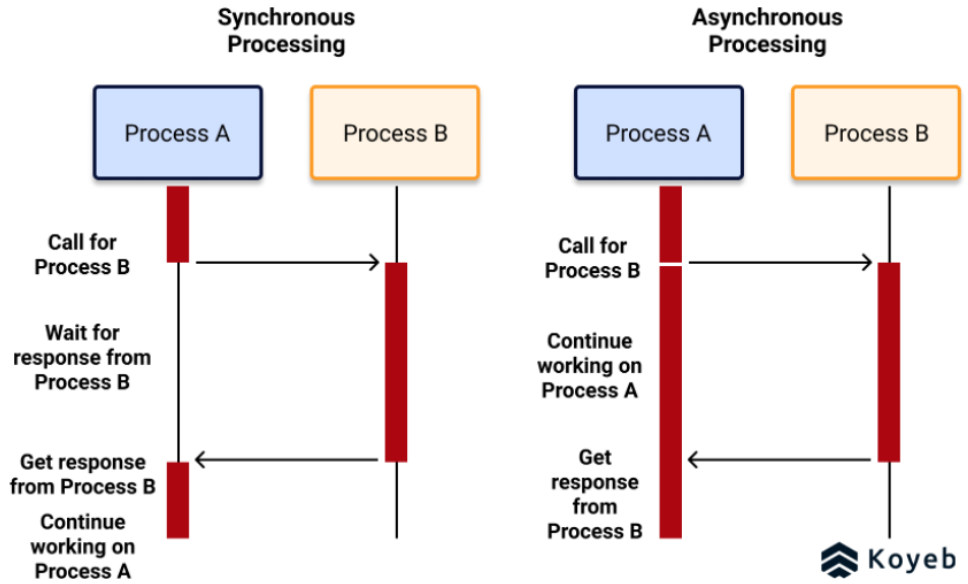
- **Synchronous function:**

Function that starts the requested task or operation and returns to the caller only after finishing its task.

- **Asynchronous function:**

Function that starts the requested task or operation and returns to the caller immediately without the result of the request, while its task is continue in the background .

Example: Starting the timer driver to count time and after that the timer informs the requester at certain time via interrupt.



Callback Function

- A callback is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time [Source : Wiki]. In simple language, If a reference of a function is passed to another function as an argument to call it, then it will be called as a Callback function.
- In C, a callback function is a function that is called through a function pointer.
- Below is a simple example to illustrate the above definition to make it more clear.

- If your application is divided into different layers. Upper layer module functions can call lower layer module functions using direct call(Call Out). On the other hand, the lower layer module functions call upper layer module functions using call back.

```c
// A simple C program to demonstrate callback
#include<stdio.h>

void A()
{
    printf("I am function A\n");
}

// callback function
void B(void (*ptr)())
{
    (*ptr) (); // callback to A
}

int main()
{
    void (*ptr)() = &A;

    // calling function B and passing
    // address of the function A as argument
    B(ptr);

return 0;
}
```

Source : https://www.geeksforgeeks.org/