35. PREPROCESSOR

The preprocessors are the directives, which give instructions to the compiler to preprocess the information before actual compilation starts. All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;). You already have seen a **#include** directive in all the examples. This macro is used to include a header file into the source file. There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc. Let us see important directives: **The #define Preprocessor** The #define preprocessor directive creates symbolic constants. The symbolic constant is called a **macro** and the general form of the directive is: `#define macro-name replacement-text` When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example: `#include <iostream> using namespace std; #define PI 3.14159 int main () { cout << "Value of PI :" << PI << endl; return 0; }` Now, let us do the preprocessing of this code to see the result assuming we have the source code file. So let us compile it with -E option and

redirect the result to

test.p. Now, if you check test.p, it will have lots of information and at the bottom, you will find the value replaced as follows: `$gcc -E test.cpp > test.p ... int main () { cout << "Value of PI :" << 3.14159 << endl; return 0; }` **Function-Like Macros** You can use #define to define a macro which will take argument as follows: `#include <iostream> using namespace std; #define MIN(a,b) (((a)<(b)) ? a : b) int main () { int i, j; i = 100; j = 30; cout <<"The minimum is " << MIN(i, j) << endl; return 0; }` If we compile and run above code, this would produce the following result: `The minimum is 30`

Conditional Compilation There are several directives, which can be used to compile selective portions of your program's source code. This process is called conditional compilation. The conditional preprocessor construct is much like the 'if' selection structure. Consider the following preprocessor code: `#ifndef NULL #define NULL 0 #endif` You can compile a program for debugging purpose. You can also turn on or off the debugging using a single macro as follows: `#ifdef DEBUG cerr <<"Variable x = " << x << endl; #endif` This causes the **cerr** statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive #ifdef DEBUG. You can use #if 0 statement to comment out a portion of the program as follows: `#if 0 code prevented from compiling #endif` Let us try the following example: #include <iostream> using namespace std; #define DEBUG #define MIN(a,b) (((a)<(b)) ? a : b) int main () { int i, j; i = 100; j = 30; #ifdef DEBUG

cerr <<"Trace: Inside main function" << endl; #endif #if 0 /* This is commented part */ cout << MKSTR(HELLO C++) << endl; #endif cout <<"The minimum is " << MIN(i, j) << endl; #ifdef DEBUG cerr <<"Trace: Coming out of main function" << endl; #endif return 0; } If we compile and run above code, this would produce the following result: Trace: Inside main function The minimum is 30 Trace: Coming out of main function **The # and Operators** The # and # # preprocessor operators are available in C++ and ANSI/ISO C. The # operator causes a replacement-text token to be converted to a string surrounded by quotes. Consider the following macro definition: #include <iostream> using namespace std; #define MKSTR(x) #x int main () { cout << MKSTR(HELLO C++) << endl;

return 0; } If we compile and run above code, this would produce the following result: HELLO C++ Let us see how it worked. It is simple to understand that the C++ preprocessor turns the line: cout << MKSTR(HELLO C++) << endl; Above line will be turned into the following line: cout << "HELLO C++" << endl; The # # operator is used to concatenate two tokens. Here is an example: #define CONCAT(x, y) x ## y When CONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, CONCAT(HELLO, C++) is replaced by "HELLO C++" in the program as follows. #include <iostream> using namespace std; #define concat(a, b) a ## b int main() { int xy = 100; cout << concat(x, y); return 0; } If we compile and run above code, this would produce the following result: 100 Let us see how it worked. It is simple to understand that the C++ preprocessor transforms:

cout << concat(x, y); Above line will be transformed into the following line: cout << xy; **Predefined C++ Macros** C++ provides a number of predefined macros mentioned below: **Macro Description** __LINE__ This contains the current line number of the program when it is being compiled. __FILE__ This contains the current file name of the program when it is being compiled. __DATE__ This contains a string of the form month/day/year that is the date of the translation of the source file into object code. __TIME__ This contains a string of the form hour:minute:second that is the time at which the program was compiled. Let us see an example for all the above macros: #include <iostream> using namespace std; int main () { cout << "Value of __LINE__ : " << __LINE__ << endl; cout << "Value of __FILE__ : " << __FILE__ << endl; cout << "Value of __DATE__ : " << __DATE__ << endl; cout << "Value of __TIME__ : " << __TIME__ << endl; return 0; }

If we compile and run above code, this would produce the following result: `Value of __LINE__ : 6 Value of __FILE__ : test.cpp Value of __DATE__ : Feb 28 2011 Value of __TIME__ : 18:52:48`

36. SIGNAL HANDLING

Signals are the interrupts delivered to a process by the operating system which can terminate a program prematurely. You can generate interrupts by pressing Ctrl+C on a UNIX, LINUX, Mac OS X or Windows system. There are signals which cannot be caught by the program but there is a following list of signals which you can catch in your program and can take appropriate actions based on the signal. These signals are defined in C++ header file <csignal>. **Signal Description** SIGABRT Abnormal termination of the program, such as a call to **abort**. SIGFPE An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow. SIGILL Detection of an illegal instruction. SIGINT Receipt of an interactive attention signal. SIGSEGV An invalid access to storage. SIGTERM A termination request sent to the program. **The signal() Function** C++ signal-handling library provides function **signal** to trap unexpected events. Following is the syntax of the signal() function: `void (*signal (int sig, void (*func)(int)))(int);` Keeping it simple, this function receives two arguments: first argument as an integer, which represents signal number and second argument as a pointer to the signal-handling function. Let us write a simple C++ program where we will catch SIGINT signal using signal() function. Whatever signal you want to catch in your program, you must register that signal using **signal** function and associate it with a signal handler. Examine the

following example:

`#include <iostream> #include <csignal> using namespace std; void signalHandler(int signum) { cout << "Interrupt signal (" << signum << ") received.\n"; // cleanup and close up stuff here // terminate program exit(signum); } int main () { // register signal SIGINT and signal handler signal(SIGINT, signalHandler); while(1){ cout << "Going to sleep...." << endl; sleep(1); } return 0; }` When the above code is compiled and executed, it produces the following result: `Going to sleep.... Going to sleep.... Going to sleep....`

Now, press Ctrl+C to interrupt the program and you will see that your program will catch the signal and would come out by printing something as follows: `Going to` sleep.... Going to sleep.... Going to sleep.... Interrupt signal (2) received. **The raise() Function** You can generate signals by function **raise()**, which takes an integer signal number as an argument and has the following syntax. int raise (signal sig); Here, **sig** is the signal number to send any of the signals: SIGINT, SIGABRT, SIGFPE, SIGILL, SIGSEGV, SIGTERM, SIGHUP. Following is the example where we raise a signal internally using raise() function as follows: #include <iostream> #include <csignal> using namespace std; void signalHandler(int signum) { cout << "Interrupt signal (" << signum << ") received.\n"; // cleanup and close up stuff here // terminate program exit(signum); } int main () { int i = 0;

// register signal SIGINT and signal handler signal(SIGINT, signalHandler); while(++i){ cout << "Going to sleep...." << endl; if(i == 3){ raise(SIGINT); } sleep(1); } return 0; } When the above code is compiled and executed, it produces the following result and would come out automatically: Going to sleep.... Going to sleep.... Going to sleep.... Interrupt signal (2) received.

37. MULTITHREADING

Multithreading is a specialized form of multitasking and a multitasking is the feature that allows your computer to run two or more programs concurrently. In general, there are two types of multitasking: process-based and thread-based. Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. C++ does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature. This tutorial assumes that you are working on Linux OS and we are going to write multi-threaded C++ program using POSIX. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris. **Creating Threads** The following routine is used to create a POSIX thread: #include <pthread.h> pthread_create (thread, attr, start_routine, arg) Here, **pthread_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of the parameters: **Parameter Description** thread An opaque, unique identifier for the new thread returned by the subroutine. attr An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values. start_routine The C++ routine that the thread will execute once it is

created. arg A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed. The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads. **Terminating Threads** There is following routine which we use to terminate a POSIX thread: `#include <pthread.h> pthread_exit (status)` Here **pthread_exit** is used to explicitly exit a thread. Typically, the pthread_exit() routine is called after a thread has completed its work and is no longer required to exist. If main() finishes before the threads it has created, and exits with pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes. **Example:** This simple example code creates 5 threads with the pthread_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit(). `#include <iostream> #include <cstdlib> #include <pthread.h> using namespace std; #define NUM_THREADS 5 void *PrintHello(void *threadid) {`

`long tid; tid = (long)threadid; cout << "Hello World! Thread ID, " << tid << endl; pthread_exit(NULL); } int main () { pthread_t threads[NUM_THREADS]; int rc; int i; for(i=0; i < NUM_THREADS; i++){ cout << "main() : creating thread, " << i << endl; rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i); if (rc){ cout << "Error:unable to create thread," << rc << endl; exit(-1); } } pthread_exit(NULL); }` Compile the following program using -lpthread library as follows: `$gcc test.cpp -lpthread` Now, execute your program which gives the following output: `main() : creating thread, 0 main() : creating thread, 1 main() : creating thread, 2 main() : creating thread, 3 main() : creating thread, 4 Hello World! Thread ID, 0 Hello World! Thread ID, 1`

`Hello World! Thread ID, 2 Hello World! Thread ID, 3 Hello World! Thread ID, 4` **Passing Arguments to Threads** This example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example: `#include <iostream> #include <cstdlib> #include <pthread.h> using namespace std; #define NUM_THREADS 5 struct thread_data{ int thread_id; char *message; }; void *PrintHello(void *threadarg) { struct thread_data *my_data; my_data = (struct thread_data *) threadarg; cout << "Thread ID : " << my_data->thread_id ; cout << " Message : " << my_data->message << endl; pthread_exit(NULL); }`

int main () { pthread_t threads[NUM_THREADS]; struct thread_data
td[NUM_THREADS]; int rc; int i; for(i=0; i < NUM_THREADS; i++){ cout
<<"main() : creating thread, " << i << endl; td[i].thread_id = i;
td[i].message = "This is message"; rc = pthread_create(&threads[i], NULL,
PrintHello, (void *)&td[i]); if (rc){ cout << "Error:unable to create
thread," << rc << endl; exit(-1); } } pthread_exit(NULL); } When the above
code is compiled and executed, it produces the following result: main() : creating
thread, 0 main() : creating thread, 1 main() : creating thread, 2 main() :
creating thread, 3 main() : creating thread, 4 Thread ID : 3 Message : This
is message Thread ID : 2 Message : This is message Thread ID : 0 Message :
This is message Thread ID : 1 Message : This is message Thread ID : 4 Message
: This is message

Joining and Detaching Threads There are following two routines which we can use to
join or detach threads: pthread_join (threadid, status) pthread_detach
(threadid) The pthread_join() subroutine blocks the calling thread until the
specified 'threadid' thread terminates. When a thread is created, one of its
attributes defines whether it is joinable or detached. Only threads that are created
as joinable can be joined. If a thread is created as detached, it can never be joined.
This example demonstrates how to wait for thread completions by using the
Pthread join routine. #include <iostream> #include <cstdlib> #include
<pthread.h> #include <unistd.h> using namespace std; #define NUM_THREADS 5
void *wait(void *t) { int i; long tid; tid = (long)t; sleep(1); cout <<
"Sleeping in thread " << endl; cout << "Thread with id : " << tid << "
...exiting " << endl; pthread_exit(NULL); } int main ()

{ int rc; int i; pthread_t threads[NUM_THREADS]; pthread_attr_t attr; void
*status; // Initialize and set thread joinable pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); for(i=0; i <
NUM_THREADS; i++){ cout << "main() : creating thread, " << i << endl; rc =
pthread_create(&threads[i], NULL, wait, (void *)i); if (rc){ cout <<
"Error:unable to create thread," << rc << endl; exit(-1); } } // free
attribute and wait for the other threads pthread_attr_destroy(&attr); for(
i=0; i < NUM_THREADS; i++){ rc = pthread_join(threads[i], &status); if (rc){
cout << "Error:unable to join," << rc << endl; exit(-1); } cout << "Main:
completed thread id :" << i ; cout << " exiting with status :" << status <<
endl; } cout << "Main: program exiting." << endl;

pthread_exit(NULL); } When the above code is compiled and executed, it produces
the following result: main() : creating thread, 0 main() : creating thread, 1
main() : creating thread, 2 main() : creating thread, 3 main() : creating
thread, 4 Sleeping in thread Thread with id : 0 .... exiting Sleeping in
thread Thread with id : 1 .... exiting Sleeping in thread Thread with id : 2
.... exiting Sleeping in thread Thread with id : 3 .... exiting Sleeping in
thread Thread with id : 4 .... exiting Main: completed thread id :0 exiting
with status :0 Main: completed thread id :1 exiting with status :0 Main:

```
completed thread id :2 exiting with status :0 Main: completed thread id :3
exiting with status :0 Main: completed thread id :4 exiting with status :0
Main: program exiting.
```