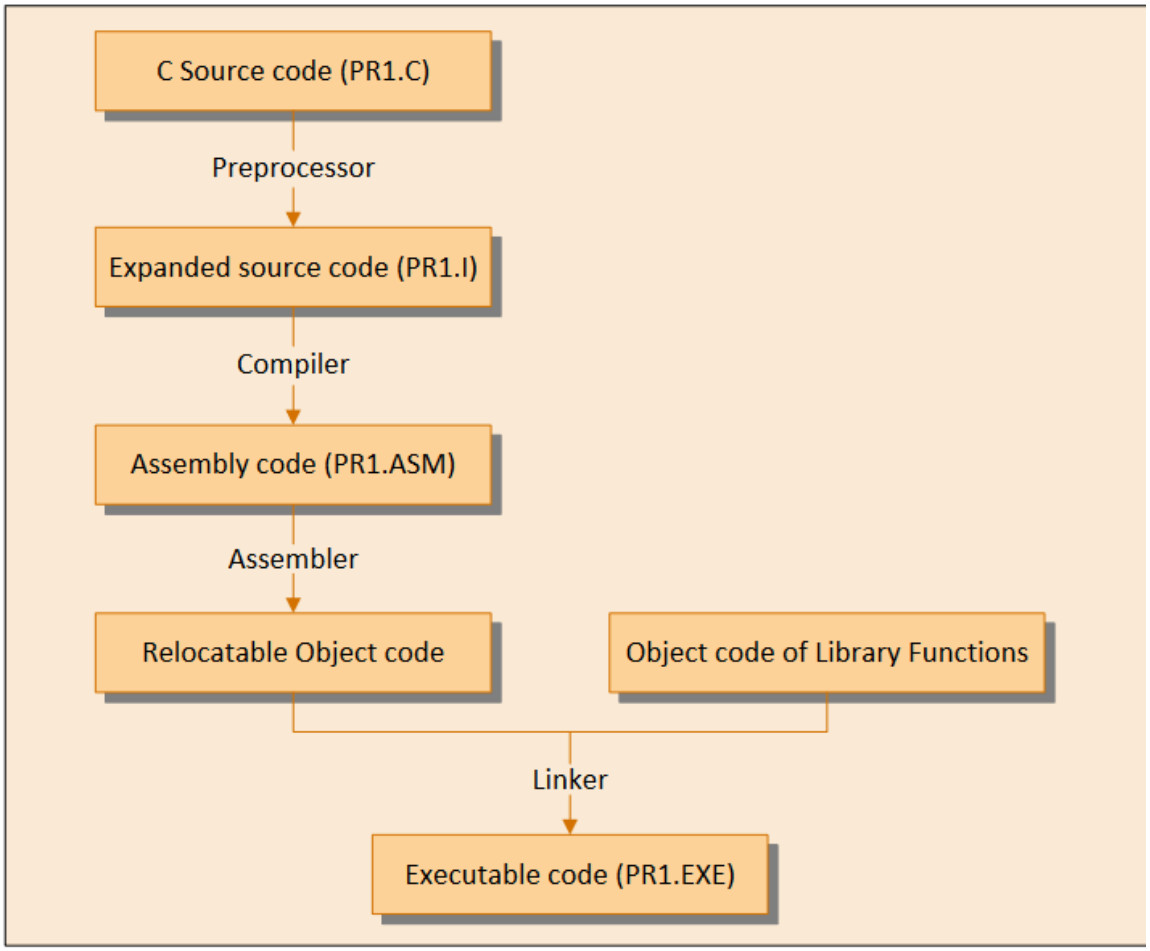
Preprocessor

It is a program that processes our source code before it is passed to the compiler.

Build process

There are many steps involved from the stage of writing a C program to the stage of getting it executed. The combination of these steps is known as the '**Build Process'**. Before a C program is compiled it is passed through another program called 'Preprocessor'.

The C program is often known as 'Source Code'. The preprocessor works on the source code and creates 'Expanded Source Code'. If the source code is stored in a file PR1.C, then the expanded source code gets stored in a file PR1.I. It is expanded source code that is sent to the compiler for compilation.



Processor	Input	Output
Editor	Program typed from keyboard	C source code containing program and preprocessor commands
Preprocessor	C Source code file	Source code file with the preprocessing commands properly sorted out
Compiler	Source code file with the preprocessing commands properly sorted out	Assembly language code
Assembler	Assembly Language code	Relocatable Object code in machine language
Linker	Object code of our program and object code of library functions	Executable code in machine language
Loader	Executable file	

Preprocessor Directives:

The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begins with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. The following preprocessor directives are:

1) Macro Expansion
2) File Inclusion
3) Conditional Compilation
4) Miscellaneous directives

1. Macro Expansion

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.

Let us now understand the macro definition with the help of a program:

```
#include<stdio.h>
#define LIMIT 20
void main()
{
int i;
for( i=1; i<=LIMIT; i++)
printf("%d\n",i);
}
```

Here

 # define LIMIT 20: this statement is called 'macro definition' or 'macro'.

LIMIT: 'macro templates'

20: 'macro expansions'

Why use macros instead of a variable

1. The compiler can generate faster and more compact code for constants than it can for variables.
2. Variable creates problem when it may inadvertently get altered somewhere in the program.

 # define directive use:

1. **#define directive is many a time used to define operators.**
   # define AND &&
   # define OR ||
2. **# define directive could be used even to replace a condition.**
   #define AND &&
   #define ARRANGE (a>25 AND a<50)
3. **# define directive could be used to replace even an entire C statement.**
   #define FOUND printf("welcome");

Macros with arguments

Macros can have argument, just as functions can.

```
#include<stdio.h>
#define AREA(x) (3.14*x*x)
void main()
{
float r1=6.25, r2=2.5,a;
 a=AREA(r1);
printf("area of circle=%f\n",a);
a=AREA(r2)
printf("Area of circle=%f\n",a);
}
```

In x in the macro template AREA(x) is an argument that matches the x in the macro expansion (3.14*x*x). The statement AREA(r1) in the program causes the variable r1 to be substituted for x. so, the statement AREA(r1) is equivalent to: (3.14*r1*r1).

Important points to remember while writing macros with arguments:

1. Not to leave a blank between the macro template and its argument while defining the macro. Eg. #define AREA(x)(3.14*x*x). no blank between AREA and x in the definition.
2. The entire macro expansion should be enclosed within parentheses.

   # define SQUARE(n) (n*n)
3. Macros can be split into multiple lines, with a '\' present at the end of each line.

   ```
   #define HLINE for(i=0;i<79;i++)\
   printf("%d",i);
   ```

Macros versus Functions

Macro	Functions
In a macro call, the preprocessor replaces the macro template with its macro expansion.	In a function call, the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.
Usually, macros make the program run faster but increase the program size.	Functions make the program smaller and compact.
Macros have already been expanded and placed in the source code before compilation.	Passing arguments to a function and getting back the returned value does take time and slow down the program.

2. **FILE Inclusion**

This directive causes one file to be included in another.

#include"filename" it simply causes the entire contents of the filename to be inserted into the source code at that point in the program. This presumes that the file being included exists.

Difference between #include"goto.h" and #include<goto.h>

#include"goto.h" : this command would look for the file goto.h in the current directory as well as the specified list of directories as mentioned in the include search path that might have been set up.

#include<goto.h> : this command would look for the file goto.h in the specified list of directories only.

/*Write macro definition with arguments for calculation of simple interest and amount. Store these macro definitions in a file called 'interest.h'. include this file in your program and use the macro definitions for calculating simple interest and amount.*/
/* for this we create file one file is interest.h and other file where we call macros*/

//interest.h file

```
#define SI(p, t, r)  ( (p * t * r) / 100.0 )
#define AMT(p, t, r) ( SI(p, t, r) + p )
```

```
// calculate simple interest and amount using macro defined in interest.h file
#include<stdio.h>
#include "interest.h"
int main()
{
   float p, t, r;
   printf("Enter principal amount\n");
   scanf("%f", &p);
   printf("Enter Rate of Interest\n");
   scanf("%f", &r);
   printf("Enter Time Period\n");
   scanf("%f", &t);
   printf("Simple Interest: %0.2f\n", SI(p, t, r));
   printf("Total Amount: %0.2f\n", AMT(p, t, r));
   return 0;
}
```

3. Conditional Compilation

Conditional Compilation directives are 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. compiler skip over part of a source code by inserting the preprocessor commands #ifdef and #endif.

Syntax:
#ifdef macroname
Statement1;
#else
Statement2;
#endif

// #ifndef (if not defined)

// program on conditional compilation

```
# include <stdio.h>
# define a 10
void main()
{
  #ifdef a
  printf("  Hello I am here..");
  #endif

  #ifndef a
  printf("  Not defined ");
  #else
  printf("  a defined ");
  #endif
}
```

Output:
Hello I am here a defined

// program on conditional compilation

```
# include <stdio.h>
void main()
{
  #ifdef a
  printf("  Hello I am here..");
  #endif

  #ifndef a
  printf("  Not defined ");
  #else
  printf("  a defined ");
  #endif
}
```

Output:

Not defined

4. Miscellanous Directives

1. **#undef** : on some occasions, it may be desirable to cause a defined name to become 'undefined'. Use #undef directive

```
#include<stdio.h>
#include <stdio.h>
#define YEARS_OLD 12
#undef YEARS_OLD

int main()
{
  #ifdef YEARS_OLD
  printf("TechOnTheNet is over %d years old.\n", YEARS_OLD);
  #endif
  printf("TechOnTheNet is a great resource.\n");
  return 0;
}
```

Output: TechOnTheNet is a great resource.

2. **#pragma** : **#pragma startup and # pragma exit** : these directives specify functions that are called upon program startup (before main() or program exit (just before the program terminates).

// this code will execute on the TC Compiler , not on GCC compiler

```
#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()

//for GCC Compiler pragma directive works as follows

```
#include<stdio.h>
void func1();
void func2();
void __attribute__((constructor)) func1();
void __attribute__((destructor)) 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()
```

2017-18 (RCS-101)

1. What is Macros? How is it substituted? Also explain macro act as a variable and macro act as a function with the help of example. 7

2017-18(RCS-201)

1. Explain the syntax and use of the following directives with examples: (i) #ifdef (II) #undef (III) #pragma (IV) # include 7

2018-19(KCS-101)

1. Write a short note on following preprocessor directives with example: (i) Macro Expansion (ii) File Inclusion 10

2018-19(KCS-201)

1. Explain the following:

   Macros (II) Union (III) Enumerated data types (iv) Type conversion 10

2019-20(KCS-101)

1. Explain the role of C preprocessor. 2
2. Write macro definition with arguments for calculation of simple interest and amount. Store this macro definition in a file called 'interest.h'. include this file in your program and use the macro definitions for calculating simple interest and amount. 10

2020-21(KCS-101T)

1. Define preprocessor and its usage in programming. 2

2021-22(KCS-101T)

1. What are header files? Why are they important. 2
2. Why are preprocessor required? Explain any two preprocessor directives.10

2021-22(KCS-201T)

1. Why are preprocessor required? Explain any two preprocessor directives.10

2022-23(BCS-101)

1. What is Macro? Illustrate the working of Macro as a variable and as a function with the help of suitable example. 7

2023-24(BCS-101)

1 Write the short notes on (i) Linked list (ii) Macros 10