Computer Science 1510

Lecture 30

Lecture Outline

- The C pre-processor
- Makefiles

The C pre-processor

- Recall that the C compilation process consists of first pre-processing the code, to create an expanded source file, which is sent to the compiler to be translated into machine language.

- We communicate with the pre-processor via source code lines beginning with a `#`.

- Thus far, we have seen only one C pre-processor directive, the `#include` directive, which is used to include files in the source code.

- A file that is included with a `#include` is copied into the source file before being sent to the compiler.

- There are several other pre-processor directives, such as `#define`, `#if`, and `#ifdef`. We will look at each of these individually.

The C pre-processor: #define

- A `#define` can be used to define a macro substitution.

- Syntax:

```
#define NAME replacement_text
```

  where every instance of `NAME` within the source code is replaced by `replacement_text` during pre-processing.

- Example:

```
#define PI 3.14159
```

- It is common to use uppercase letters for defined elements to distinguish them as text that will be replaced.

- The scope of a `#define` is from the point of the `#define` to the end of the source file.

- To continue a macro onto more than one line a \ should be placed at the end of each line to be continued.

The C pre-processor: #define

- A macro may use previous macros.
- A macro can have arguments, such that the `replacement_text` may differ between calls to the macro.
- Example:

```
#define CIRC(R) 2*PI*(R)
```

- Although the above looks like a function call, it is still simply a replacement of text. For example,

```
c=CIRC(2*r+3);
```

  expands to `c=2*3.14159*(2*r+3);` during pre-processing.

- Note that the parenthesis are important here, to prevent `c=2*2.14159*2*r+3`

The C pre-processor: #define

- A `NAME` in a string is not replaced. For example, in

```
printf("CIRC\n");
```

  `CIRC` would not be replaced.

- It is also possible to undefine a macro with the `#undef` directive.

- For example,

```
#undef PI
```

  will undefine the substitution macro `PI`.

The C pre-processor: #define

- Another example of macro substitution this evaluates the greater of the two arguments:

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

- Although this looks a bit like a function, it is different in several ways:
  - There is no function call, the macro is instantiated in place in the program. (This may be more efficient for small functions.)
  - No type checking is done, so this will work for any compatible types.

- These may be both advantages and disadvantages – use macros carefully and sparingly.

The C pre-processor: #if

- The pre-processor can be used for conditional inclusion of different portions of a source file.

- Syntax:

```
#if expression1
   statements
#elif expression2
   statements
#else
   statements
#endif
```

  where the different # lines behave as a standard if, else if, else block.

- Each expression must evaluate to an integer, which is treated as true for nonzero values, and false for a zero value.

The C pre-processor: #if

- This can be useful when debugging source code, in particular, to comment out a large block of code. (Note that C comments /* */ do not nest).

- Example:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   declarations;
   statements1;
   #if 0
   statements2;
   #endif
   statements3;
   return 0;
}
```

- In the above example, `statements2` would not be included in the compiled version, it would be removed by the pre-processor since the if expression is equal to 0.

The C pre-processor: #ifdef, #ifndef

- The `#ifdef` directive is a specialized form of `#if` that is used to test if something has already been defined. `#ifndef` tests if something has not already been defined.

- Syntax:

```
#ifndef NAME
```

  `#ifndef` is ended with an `#endif`.

- `#ifdef` has similar syntax.

- This is often used in header files to ensure that function definitions have not been included more than once.

- For example, we could have a header file that contains the following:

```
#ifndef FACTORIAL
#define FACTORIAL
int factorial(int n);
#endif
```

The C pre-processor: #ifdef, #ifndef

- It is good programming practice to use the above structure in all header files.

- On the first inclusion of the given header file, `FACTORIAL` is not yet defined, resulting in the execution of the second and third lines (up to the `#endif`).

- Now `FACTORIAL` is defined, and the `factorial` function is declared.

- Thus, any subsequent inclusions of this header file will see that `FACTORIAL` has already been defined, and will not attempt to redeclare the function.

The C pre-processor: #if

- We could also use a `#define` to remove sections of code for debugging purposes.

- Example:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
   declarations;
   statements1;
   #ifdef DEBUG
   statements2;
   #endif
   statements3;
   return 0;
}
```

- In the above example, `statements2` would be included in the compiled version only if `DEBUG` is defined.

- `DEBUG` can be defined by,

  1. including a `#define DEBUG` within the code, or,
  2. using the `-D` flag during compilation. For example,

     ```
     gcc -Wall -DDEBUG file.c
     ```

Makefiles

- We saw in lab 8 how to create and compile programs using multiple files.

- Rather than having to manually compile each of the files containing functions, to object files, followed by linking the .o files into the file containing the main function, we can use what is called a *Makefile*.

- A Makefile describes which files should be compiled to object files, and which object files each .c file depends on.

- To compile the program using a Makefile we would simply type `make`.

- A Makefile is placed in a file called `Makefile`, and contains *targets* followed by a list of files and compilation commands required to build that target.

Example 1: Makefile

```
#!/bin/bash

binomial: factorial.o bicoeff.o bicoeff.h
   gcc -Wall -o binomial binomial.c bicoeff.o factorial.o

factorial.o: factorial.c
   gcc -Wall -c factorial.c

bicoeff.o: bicoeff.c factorial.h
   gcc -Wall -c bicoeff.c

clean:
   rm binomial bicoeff.o factorial.o
```

Example 2: Makefile

```
#!/bin/bash

OBJS = bicoeff.o factorial.o
CC = gcc
FLAGS = -Wall

binomial: $(OBJS) bicoeff.h
   $(CC) $(FLAGS) -o binomial binomial.c $(OBJS)

factorial.o: factorial.c
   $(CC) $(FLAGS) -c factorial.c

bicoeff.o: bicoeff.c factorial.h
   $(CC) $(FLAGS) -c bicoeff.c

clean:
   rm binomial $(OBJS)
```

Example: Linked list

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int value;
   struct node *next;
};

int main(int argc, char *argv[]) {
   int i, ret=0;
   struct node *top, *cur;

   top=NULL;  /* Initialize top */
   /* Read in node values */
   printf("Please enter a value (CTRL^D to stop)\n");
   ret = scanf("%d",&i);
   while (ret != EOF) {   /* Until EOF is read */
      if (top==NULL) {   /* The list has not been started */
         /* Allocate space for first node */
         top = (struct node*)malloc(sizeof(struct node));
         cur=top;  /* Point cur to the start of the list */
      } else {   /* Add a new node to the end of the list */
         cur->next = (struct node*)malloc(sizeof(struct node));
         cur=cur->next;  /* Point cur to this new node */
      }
      cur->next = NULL;  /* next does not yet point to anything */
      cur->value=i;      /* Set the element's value to the value read */
      printf("Please enter a value (CTRL^D to stop)\n");
      ret = scanf("%d",&i);
   }

   if (top==NULL) {
      printf("No data read\n");
      return 0;
   }
```

```
   printf("The values are:\n");
   cur=top;  /* Start at the first node in the list */
   while (cur!=NULL) {
      printf("%d\n",cur->value);
      cur = cur->next;  /* Move to the next node in the list */
   }

   return 0;
}
```