CSC 270 – Survey of Programming Languages

C Lecture 7 – Separate Compilation

Why Compile Separately

- There are several advantages to being able to break larger programs into separate modules:
  - It takes less time to compile the few procedures that have been changed and then link them to the rest of the program.
  - It makes it easier to write a useful module once and then link it to the rest of the program.

Example – Using a stack

- A stack is a data structure where items are inserted using **`pop()`** and removed using **`push()`**. We can see if there is anything in it using **`empty()`**.
- A stack is a last-in-first-out data structure.
- The first version of our program contains both the data items that the stack needs as well as its procedures.

StackStuf.c

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

#define           STACKSIZE   100
typedef     int   StackItem;

typedef     struct      {
      StackItem   item[STACKSIZE];
      int   top;
}     Stack;

Stack s;
```

```
/* The functions that manipulate the stack */
void        error(char *message);
void        stackInit(Stack *s);
int         stackEmpty(Stack s);
StackItem   stackPop(Stack *s);
void        stackPush(Stack *s, StackItem x);

/*
 * main() - A driver to demonstate the stack
 */
int main(void)
{
      int   x;
```

```
      /* Initialize the stack, setting top to 0 */
      stackInit(&s);
      do    {
            printf
               ("Enter an integer; 0 to quit\t?");
            scanf("%d", &x);

            if (x != 0)
                  stackPush(&s, x);
      } while (x != 0);
```

```
        /*
         * Print everything on the stack from the top
         * down
         */
        while (!stackEmpty(s))  {
                x = stackPop(&s);
                printf("Top was %d\n", x);
        }
        return(0);
}
```

```
/*
 * stackInit() - initialize the top as 0
 */
void  stackInit(Stack *s)
{
        s->top = 0;
}

/*
 * stackEmpty() - Returns true if the stack is
 *                              empty; false if not
 */
int   stackEmpty(Stack s)
{
        return(s.top == 0);
}
```

```
/*
 * StackPop() - Pops an item from the top of the
 *                      stack
 */
StackItem   stackPop(Stack *s)
{
      if (stackEmpty(*s))
            error("Stack underflow");
      else
            return(s->item[--s->top]);
}
```

```
/*
 * stackPush() - Pushes an item on the stack
 */
void        stackPush(Stack *s, StackItem x)
{
      if (s->top == STACKSIZE-1)
            error("Stack overflow");
      else
            s->item[s->top++] = x;
}
```

```
/*
 * error() -      Prints an error message and
 *                      terminates execution
 */
void        error(char *message)
{
            printf("%s\n", message);
            exit(1);
}
```

Compiling the Stack Separately

- We can't use the stack without first declaring the stack's structure and providing prototypes for its procedures.
- We place these in a header file and we can include it by giving it the extension "**`.h`**"
- We enclose the file's name in quotation marks to indicate that it is in the same directory as the other source code files.

StackStuf2.c

```
#include    <stdio.h>
#include    "Stack.h"

/*
 * main() - A driver to demonstate the stack
 */
int main(void)
{
      … …
}
```

#ifndef

- The file containing the stack's procedures as well as any file using the stack will need to include **`stack.h`**, so there is a chance that we may appear to declare more than once.
- To avoid this problem, we use the conditional preprocessor **`#ifndef`**, which only includes the source code following it if the data item appearing on that line has been defined.
- **`#ifdef`**, its opposite only includes the source code if the item IS defined.

#define

- **#define** *identifier* can be used to define an identifier, even if we do not give it a value.
- Example

```
#ifndef  NAME_H
#define  NAME_H
… …
#endif
```

Stack.h

```
#ifndef     STACK_H
#define     STACK_H
#include    <stdio.h>
#include    <stdlib.h>

#define           STACKSIZE   100
typedef     int   StackItem;

typedef     struct      {
      StackItem   item[STACKSIZE];
      int   top;
}     Stack;

Stack s;
```

```
/* The functions that manipulate the stack */
void        error(char *message);
void        stackInit(Stack *s);
int               stackEmpty(Stack s);
StackItem   stackPop(Stack *s);
void        stackPush(Stack *s, StackItem x);

#endif
```

Stack.c

```
#include    "Stack.h"
void  stackInit(Stack *s)
{
      … …
}
int   stackEmpty(Stack s)
{
      … …
}
StackItem   stackPop(Stack *s)
{
      … …
}
```

```
void        stackPush(Stack *s, StackItem x)
{
      … …
}

void        error(char *message)
{
      … …
}
```

Example – A Point

- Imagine that we wish to describe a point on a graph or a vector value (a value having both magnitude and direction).
- We need to store both x- and y-coordinates.
- We need to be able to read, write, add and subtract vectors as well as determine its distance from the origin.

Point.h

```
#ifndef POINT_H
#define POINT_H

/* The coordinates of a point */
typedef     struct      {
      int   x, y;
} Point;

/* Input and output functions */
void  readPoint(Point *p);
void  writePoint(Point p);
```

```
/* These two functions treat p and q as vectors */
/* Returns p + q */
Point addPoints(Point p, Point q);

/* Return p - q */
Point subPoints(Point p, Point q);
float distance(Point p);

#endif /* not defined POINT_H */
```

Point.cpp

```
#include    "Point.h"
#include    <stdio.h>
#include    <math.h>

void  readPoint(Point *p)
{
      printf("Enter x\t?");
      scanf("%d", &(p->x));
      printf("Enter y\t?");
      scanf("%d", &(p->y));
}
```

```
#include    "Point.h"
#include    <stdio.h>
#include    <math.h>

/*
 * readPoint() - reads two coordinates of a point
 */
void  readPoint(Point *p)
{
      printf("Enter x\t?");
      scanf("%d", &(p->x));
      printf("Enter y\t?");
      scanf("%d", &(p->y));
}
```

```
/*
 * writePoint() - Writes a point's coordinates in
 *                (x, y) format
 */
void  writePoint(Point p)
{
      printf("(%d, %d)", p.x, p.y);
}
```

```
/*
 * addPoints() - Returns p + q
 */
Point addPoints(Point p, Point q)
{
      Point r;

      r.x = p.x + q.x;
      r.y = p.y + q.y;
      return(r);
}
```

```
/*
 * subPoints() - Returns p - q
 */
Point subPoints(Point p, Point q)
{
      Point r;

      r.x = p.x - q.x;
      r.y = p.y - q.y;
      return(r);
}
```

```
/*
 * distance() - Returns p's distance from the origin
 */
float       distance(Point p)
{
      return(sqrt((float)(p.x*p.x + p.y*p.y)));
}
```

main.c

```
#include    "Point.h"
#include    <stdio.h>

int   main(void)
{
      Point p, q, r;
      float dist;

      readPoint(&p);
      readPoint(&q);

      r = addPoints(p, q);
      printf("p + q = ");
```

```
      writePoint(r);
      printf("\n");

      r = subPoints(p, q);
      printf("p - q = ");
      writePoint(r);
      printf("\n");

      dist = distance(p);
      printf("dist = %f\n", dist);


      return(0);
}
```