ESc 101: Fundamentals of Computing

Lecture 32

Mar 31, 2010

Outline

1 Fibonacci Numbers

2 Allocating Memory Dynamically

Computing Large Fibonacci Numbers

- To store large Fibonacci numbers, an `int` type variable in insufficient.
- Instead, we can use functions for adding large numbers developed in the beginning of the course.
- A number is now stored as an array of `SIZE+1` characters with last symbol storing the sign.

Computing Large Fibonacci Numbers

- To store large Fibonacci numbers, an `int` type variable in insufficient.
- Instead, we can use functions for adding large numbers developed in the beginning of the course.
- A number is now stored as an array of `SIZE+1` characters with last symbol storing the sign.

Computing Large Fibonacci Numbers

- To store large Fibonacci numbers, an `int` type variable in insufficient.
- Instead, we can use functions for adding large numbers developed in the beginning of the course.
- A number is now stored as an array of `SIZE+1` characters with last symbol storing the sign.

Function Fib_loop()

```
/* Computes the nth Fibonacci number and stores it
 * in num.
 */
void Fib_loop(int n, char num[])
{
    char F[N][SIZE+1];

    set_number(F[0], 1); // set first two numbers
    set_number(F[1], 1);

    for (int m = 2; m <= n; m++)
        add_numbers(F[m-2], F[m-1], F[m]);

    copy_number(num, F[n]);
}
```

Defining New Types

- Instead of defining a long integer everywhere as a character array, it would be much nicer if we can define our own type of variable, say `Number`.
- It is possible using `typedef` command.
- We can, for example, say:
  `typedef char Number[SIZE+1];`
- This defines a type called `Number`, which is same as an array of `SIZE+1` symbols.
- Now, everywhere we can define number variables as of type `Number`.
- A new type is defined exactly as a variable name.

Defining New Types

- Instead of defining a long integer everywhere as a character array, it would be much nicer if we can define our own type of variable, say `Number`.
- It is possible using `typedef` command.
- We can, for example, say:
  `typedef char Number[SIZE+1];`
- This defines a type called `Number`, which is same as an array of `SIZE+1` symbols.
- Now, everywhere we can define number variables as of type `Number`.
- A new type is defined exactly as a variable name.

Defining New Types

- Instead of defining a long integer everywhere as a character array, it would be much nicer if we can define our own type of variable, say `Number`.
- It is possible using `typedef` command.
- We can, for example, say:
  ```
  typedef char Number[SIZE+1];
  ```
- This defines a type called `Number`, which is same as an array of `SIZE+1` symbols.
- Now, everywhere we can define number variables as of type `Number`.
- A new type is defined exactly as a variable name.

Defining New Types

- Instead of defining a long integer everywhere as a character array, it would be much nicer if we can define our own type of variable, say `Number`.
- It is possible using `typedef` command.
- We can, for example, say:
  `typedef char Number[SIZE+1];`
- This defines a type called `Number`, which is same as an array of `SIZE+1` symbols.
- Now, everywhere we can define number variables as of type `Number`.
- A new type is defined exactly as a variable name.

Defining New Types

- Instead of defining a long integer everywhere as a character array, it would be much nicer if we can define our own type of variable, say `Number`.
- It is possible using `typedef` command.
- We can, for example, say:
  `typedef char Number[SIZE+1];`
- This defines a type called `Number`, which is same as an array of `SIZE+1` symbols.
- Now, everywhere we can define number variables as of type `Number`.
- A new type is defined exactly as a variable name.

Defining New Types

- Instead of defining a long integer everywhere as a character array, it would be much nicer if we can define our own type of variable, say `Number`.
- It is possible using `typedef` command.
- We can, for example, say:
  `typedef char Number[SIZE+1];`
- This defines a type called `Number`, which is same as an array of `SIZE+1` symbols.
- Now, everywhere we can define number variables as of type `Number`.
- A new type is defined exactly as a variable name.

Function Fib_loop() Again

```
typedef char Number[SIZE+1];

/* Computes the nth Fibonacci number and stores it
 * in num.
 */
void Fib_loop(int n, Number num)
{
    Number F[N];

    set_number(F[0], 1); // set first two numbers
    set_number(F[1], 1);

    for (int m = 2; m <= n; m++)
        add_numbers(F[m-2], F[m-1], F[m]);

    copy_number(num, F[n]);
}
```

Changing Addition Functions

- We rewrite all functions using the new type.
- Create a header file and put all function declarations and type definition there.
- Split the functions in two files: one for I/O and one for addition.

Changing Addition Functions

- We rewrite all functions using the new type.
- Create a header file and put all function declarations and type definition there.
- Split the functions in two files: one for I/O and one for addition.

Changing Addition Functions

- We rewrite all functions using the new type.
- Create a header file and put all function declarations and type definition there.
- Split the functions in two files: one for I/O and one for addition.

Outline

1 Fibonacci Numbers

2 Allocating Memory Dynamically

Runtime Allocation of Memory

- So far, all our variables have been allocated space by the compiler, and the space is fixed during the execution of the program.
- This means that the space allocated to an array is fixed a-priory, irrespective of whether during the execution of the program, less or more is actually needed.
- For example, for storing large numbers, we have fixed the number of digits to `SIZE`.
- This is inconvenient since, depending on where the library for addition is used, the size requirements may be different.
- C provides a way to handle this: by allocating memory at the time of execution instead of at the time of compilation.

Runtime Allocation of Memory

- So far, all our variables have been allocated space by the compiler, and the space is fixed during the execution of the program.
- This means that the space allocated to an array is fixed a-priory, irrespective of whether during the execution of the program, less or more is actually needed.
- For example, for storing large numbers, we have fixed the number of digits to `SIZE`.
- This is inconvenient since, depending on where the library for addition is used, the size requirements may be different.
- C provides a way to handle this: by allocating memory at the time of execution instead of at the time of compilation.

Runtime Allocation of Memory

- So far, all our variables have been allocated space by the compiler, and the space is fixed during the execution of the program.
- This means that the space allocated to an array is fixed a-priory, irrespective of whether during the execution of the program, less or more is actually needed.
- For example, for storing large numbers, we have fixed the number of digits to `SIZE`.
- This is inconvenient since, depending on where the library for addition is used, the size requirements may be different.
- C provides a way to handle this: by allocating memory at the time of execution instead of at the time of compilation.

Runtime Allocation of Memory

- So far, all our variables have been allocated space by the compiler, and the space is fixed during the execution of the program.
- This means that the space allocated to an array is fixed a-priory, irrespective of whether during the execution of the program, less or more is actually needed.
- For example, for storing large numbers, we have fixed the number of digits to `SIZE`.
- This is inconvenient since, depending on where the library for addition is used, the size requirements may be different.
- C provides a way to handle this: by allocating memory at the time of execution instead of at the time of compilation.

Runtime Allocation of Memory

- So far, all our variables have been allocated space by the compiler, and the space is fixed during the execution of the program.
- This means that the space allocated to an array is fixed a-priory, irrespective of whether during the execution of the program, less or more is actually needed.
- For example, for storing large numbers, we have fixed the number of digits to `SIZE`.
- This is inconvenient since, depending on where the library for addition is used, the size requirements may be different.
- C provides a way to handle this: by allocating memory at the time of execution instead of at the time of compilation.

The `malloc()` Function

- `malloc(n)` allocates a contiguous memory block of size `n` bytes and returns a pointer to the first byte.
- `free(p)` deallocates the memory block pointed to by `p`.

The `malloc()` Function

- `malloc(n)` allocates a contiguous memory block of size `n` bytes and returns a pointer to the first byte.
- `free(p)` deallocates the memory block pointed to by `p`.

Function Fib_loop() Yet Again

```
typedef char *Number;
int SIZE = 10; // Represents the number of digits in a number

/* Computes the nth Fibonacci number and stores it
 * in num.
 */
void Fib_loop(int n, Number num)
{
    Number F[N];

    for (int m = 0; m <= n; m++) // allocate space
        F[m] = (Number) malloc(SIZE); // Value of SIZE to be r

    set_number(F[0], 1); // set first two numbers
    set_number(F[1], 1);

    for (int m = 2; m <= n; m++)
```