Chapter - 23
Modular Programming

Modules

A module is a collection of functions or classes that perform related functions.
If a program is a book, a module is a chapter.
Modules consist of a public and private part.

Public Part

- Defines how the module is to be used
- Is put in a header file to be used by other people

Private Part

- Does the work
- Contains the “details” that the user does not have to worry about.
- Is put in a C++ source file.

extern modifier

Is used to indicate a function or variable defined in another file.

File: main.cpp

```
#include <iostream>
/* number of times through the loop */
extern int counter;

/* routine to increment the counter */
extern void inc_counter(void);

int main() {
    int   index; /* loop index */

    for (index = 0; index < 10; ++index)
        inc_counter();
    std::cout << "Counter is " << counter << '\n';
    return (0);
}
```

extern modifier

Note: No modifier such as "static".
This indicates a public variable which
can be used (if extern declared) in another file

File: count.cpp

```
/* number of times through the loop */
int counter = 0;


/* trivial example */
void inc_counter() {
    ++counter;
}
```

No modifier.

Modifiers for Global Data

Modifier	Meaning
`extern`	Variable/function is defined in another file.
<blank>	Variable/function is defined in this file (public) and can be used in other files.
`static`	Variable/function is local to this file (private).

Examples

The following is legal:

```
extern sam;
int sam = 1;    // this is legal
```

The following is legal, but causes a lot of problems. (Some linkers check for this and scream when the see it.)

File: main.cpp

```
int flag  = 0;      // flag is off

main(){
    std::cout << "Flag is " << flag << '\n';
    // ....
```

File: sub.cpp

```
int flag = 1;       // flag is on
```

static is OK

File: main.cpp

```
static int      flag  = 0;      // flag is off

int main() {
    std::cout << "Flag is " << flag << '\n';
}
```

File: sub.cpp

```
static int      flag = 1;       // flag is on
```

Headers, the public part

Headers contain:

- A comment section describing clearly what the module does and what is available to the user.
- Public class definitions
- Common constants.
- Public structures.
- Prototypes of all the public functions.
- extern declarations for public variables.

File: ia.h

```
/********************************************************
 * definitions for the infinite array (ia) class        *
 *                                                      *
 * An infinite array is an array whose size can grow    *
 * as needed.  Adding more elements to the array        *
 * will just cause it to grow.                          *
 *------------------------------------------------------*
 * class infinite_array                                 *
 *    Member functions                                  *
 *      infinite_array(void)  -- default constructor    *
 *      ~infinite_array(void) -- destructor             *
 *      int &operator [](int index)                     *
 *              gets an element of the infinite array   *
 ********************************************************/

// number of elements to store in each cell of the inf. array
const unsigned int BLOCK_SIZE = 100;
```

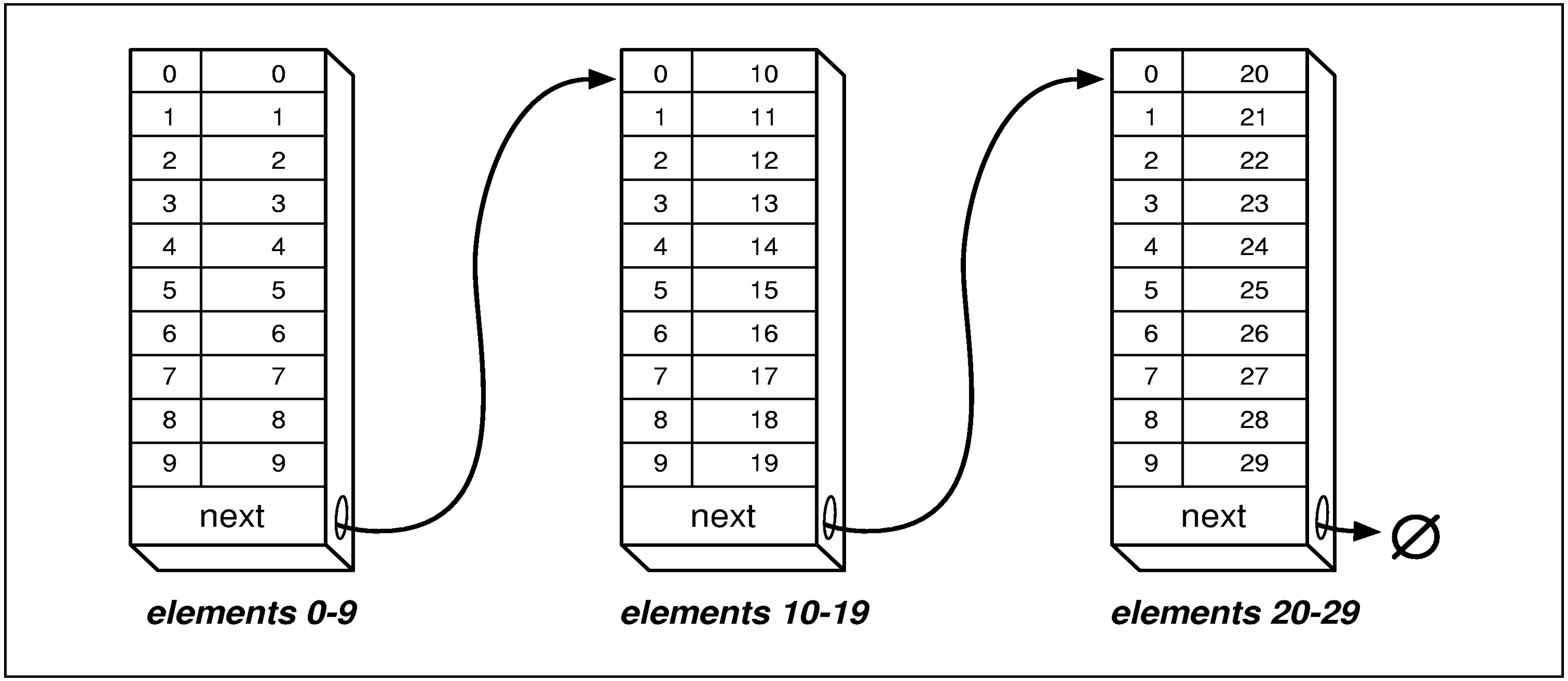
File: ia.h (continued)

```
class infinite_array {
    private:
        // the data for this block
        int   data[BLOCK_SIZE];

        // pointer to the next array
        class infinite_array *next;
    public:
        // Default constructor
        infinite_array(void)
        {
            next = NULL;
            memset(data, '\0', sizeof(data));
        }

        // Default destructor
        ~infinite_array(void);

        // Return a reference to an element of the array
        int &operator[] (const unsigned int index);
};
```

Infinite Array Code

Note: In order to use the Infinite array, all you need is the header. The real work is done in the body of the program which you don't need to see.
All functions and variables in the body that are not public are declared static to hide them from the outside world.

Infinite array structure



ia.cpp

```
/********************************************************
 * infinite-array -- routines to handle infinite arrays *
 *                                                      *
 * An infinite array is an array that grows as needed.  *
 * There is no index too large for an infinite array    *
 * (unless we run out of memory).                       *
 ********************************************************/
#include <iostream>
#include <stdlib.h>
#include <string.h>


#include "ia.h"                 // get common definitions
```

ia.cpp (continued)

```
/*********************************************************
 * operator [] -- find an element of an infinite array  *
 *                                                       *
 * Parameters                                            *
 *      index   -- index into the array                  *
 *                                                       *
 * Returns                                               *
 *      Reference to the element in the array            *
 *********************************************************/
```

ia.cpp (continued)

```
int &infinite_array::operator [] (const unsigned int index){
    // pointer to the current bucket
    class infinite_array *current_ptr;
    int current_index;  // Index we are working with
    current_ptr = this;
    current_index = index;
    while (current_index >= BLOCK_SIZE) {
        if (current_ptr->next == NULL) {
            current_ptr->next = new infinite_array;
            if (current_ptr->next == NULL) {
                cerr << "Error:Out of memory\n";
                exit(8);
            }
        }
        current_ptr = current_ptr->next;
        current_index -= BLOCK_SIZE;
    }
    return (current_ptr->data[current_index]);
}
```

ia.cpp (continued)

```
/*********************************************************
 * ~infinite_array -- Destroy the infinite array        *
 *********************************************************/
infinite_array::~infinite_array(void)
{
    /*
     * Note: We use a cute trick here.
     *
     * Because each bucket in the infinite array is
     * an infinite array itself, when we destroy
     * next, it will destroy all that bucket's "next"s,
     * and so on, recusively clearing the entire array.
     */
    if (next != NULL) {
       delete next;
       next = NULL;
    }
}
```

Makefile for GNU g++ command

```
# Make file needs debugging
CFLAGS = -g -Wall
SRC=ia.cc hist.cc
OBJ=ia.o  hist.o


all: hist


hist: $(OBJ)
        g++ $(CFLAGS) -o hist $(OBJ)


hist.o:ia.h hist.cc
        g++ $(CFLAGS) -c hist.cc


ia.o:ia.h ia.cc
        g++ $(CFLAGS) -c ia.cc


clean:
        rm hist io.o hist.o
```

Using the Infinite array

File: hist.cpp

```
/********************************************************
 * hist -- generate a histogram of an array of numbers  *
 *                                                      *
 * Usage                                                *
 *      hist <file>                                     *
 *                                                      *
 * Where                                                *
 *      file is the name of the file to work on         *
 ********************************************************/
#include <iostream>
#include <fstream.h>
#include <iomanip.h>


#include <stdlib.h>
#include <string.h>


#include "ia.h"
```

hist.cpp (continued)

```
/*
 * the following definitions define the histogram
 */
const int NUMBER_OF_LINES = 50; // # Lines in the result
const int LOW_BOUND       = 0;  // Lowest number we record
const int HIGH_BOUND      = 99; // Highest number we record
/*
 * if we have NUMBER_OF_LINES data to
 * output then each item must use
 * the following factor
 */
const int FACTOR =
  ((HIGH_BOUND - LOW_BOUND +1) / NUMBER_OF_LINES);

// number of characters wide to make the histogram
const int WIDTH = 60;

// Array to store the data in
static infinite_array data_array;
// Number if items in the array
static int data_items;
```

hist.cpp (continue)

```
int main(int argc, char *argv[])
{
    void  read_data(char *name);// get the data in
    void  print_histogram(void);// print the data

    if (argc != 2) {
        std::cerr << "Error:Wrong number of arguments\n";
        std::cerr << "Usage is:\n";
        std::cerr << "  hist <data-file>\n";
        exit(8);
    }
    data_items = 0;

    read_data(argv[1]);
    print_histogram();
    return (0);
}
```

```
/********************************************************
 * read_data -- read data from the input file into      *
 *              the data_array.                         *
 *                                                      *
 * Parameters                                           *
 *      name -- the name of the file to read            *
 ********************************************************/
void  read_data(char *name)
{
    ifstream in_file(name); // input file
    int data;               // data from input

    if (in_file.bad()) {
        cerr << "Error:Unable to open " << name << '\n';
        exit(8);
    }
    while (!in_file.eof()) {
        in_file >> data;

        // If we get an eof we ran out of data in last read
        if (in_file.eof())
           break;

        data_array[data_items] = data;
        ++data_items;
    }
}
```

hist.cpp (continued)

```
/********************************************************
 * print_histogram -- print the histogram output.       *
 ********************************************************/
void  print_histogram(void)
{
    // upper bound for printout
    int   counters[NUMBER_OF_LINES];
    int low;            // lower bound for printout
    int   out_of_range = 0;// number of items out of bounds
    int   max_count = 0;// biggest counter
    float scale;        // scale for outputting dots
    int   index;        // index into the data


    memset(counters, '\0', sizeof(counters));
```

hist.cpp (continued)

```
for (index = 0; index < data_items; ++index) {
    int data;// data for this point


    data = data_array[index];


    if ((data < LOW_BOUND) || (data > HIGH_BOUND))
        ++out_of_range;
    else {
        // index into counters array
        int   count_index;


        count_index =
               int (float(data - LOW_BOUND) / FACTOR);


        ++counters[count_index];
        if (counters[count_index] > max_count)
            max_count = counters[count_index];
    }
}
```

hist.cpp (continued)

```
    scale = float(max_count) / float(WIDTH);

    low = LOW_BOUND;

    for (index = 0; index < NUMBER_OF_LINES; ++index) {
        // index for outputting the dots
        int   char_index;
        int   number_of_dots;   // number of * to output

        std::cout << setw(2) << index << ' ' <<
                setw(3) << low << "-" <<
                setw(3) << low + FACTOR -1 << " (" <<
                setw(4) << counters[index] << "): ";

        number_of_dots = int(float(counters[index]) / scale);
        for (char_index = 0; char_index < number_of_dots;
             ++char_index)
            std::cout << '*';
        std::cout << '\n';
        low += FACTOR;
    }
    std::cout << out_of_range << " items out of range\n";
}
```