Consider a program that is required to find all values larger than the average in a list of integers. The list is stored in a file. The program must read and store the list to fulfill its requirement. The question is "How big is the biggest list that program will ever be required to process?" The answer is the size of the list could vary by orders of magnitude. Allocating storage for the largest list is a profligate waste of resource for most invocations of the program. The solution is to allocate only as much storage as is required. Programming languages have a facility that accommodates the need to allocate storage whose size can only be determined during execution. The solution is dynamic memory allocation.

Allocating and reclaiming storage for scalars: Providing programming constructs that allocate and reclaim storage fulfills the requirement. The construct for allocating storage in C++ is **new**. The construct that reclaims storage is **delete**.

Program **p01** allocates storage for an integer, assigns the integer a value of five (5), prints the integer, and reclaims storage for the integer.

```
#include <iostream>
using namespace std;
int main()
{    int* p;
     p=new int;
     *p=5;
     cout << endl;
     cout << "p=" << p;
     cout << endl;
     cout << "*p=" << *p;
     cout << endl;
     delete p;
     return 0;
}
```

Figure 1. Program **p01**.

Program **p01** prints:

```
p=00301E50
*p=5
```

Let us consider variable *p*. Variable *p* is a pointer. Variable p points to unnamed storage for an integer as illustrated in Figure 2.

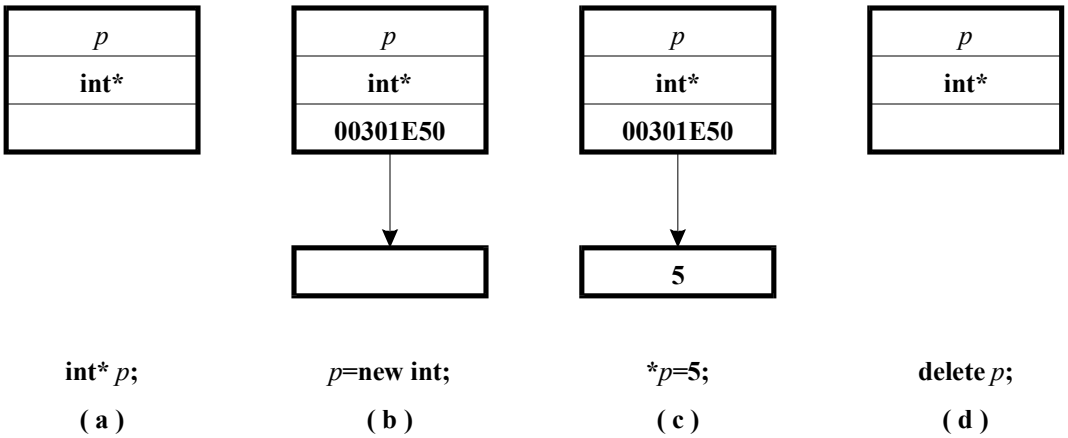

Figure 2. Variable *p*.

Storage for variable *p* is allocated when it is declared as shown in Figure 2(a). The empty space in the bottom box of Figure 2(a) signifies that value of variable *p* is undefined.

Storage for an unnamed integer is allocated in Figure 2(b). Note that variable *p* is assigned a value. The value assigned to variable *p* is the *address* of the storage for the unnamed integer. Executing program p01 will likely produce different addresses for *p*.

In Figure 2(c), the unnamed integer is assigned the integer value **5**. Note the use of the asterisk (*****). The left-hand side of the assignment expression is read from right to left. The left-hand side of the assignment expression ***p** is read "obtain the value of *p*, now assign the value to the address stored as a value in *p*."

Finally, in Figure 2(d) storage for the unnamed integer is reclaimed and the value of *p* is undefined.

Allocating and reclaiming storage for arrays: Storage for arrays may be allocated and reclaimed using a slight variation of the technique for allocating and reclaiming storage for scalars. An array of one hundred integers may be allocated in the following declaration and initialization.

int* *L***=new int[100];**

Storage referenced by variable *L* may be reclaimed as shown below.

delete[] *L***;**

Note that variable *L* is a pointer to an integer. Variable L points to the first element of the integer array. Pictorially, variable *L* is shown in Figure 3.

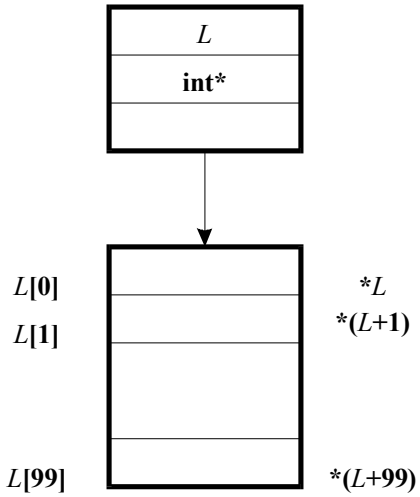

Figure 3. Variable *L*

Referencing elements of the anonymous array pointed to by variable *L* is easy. Reference elements of the array as if variable L were an integer array. Note the equivalent element reference syntax on either side of array elements. Element 1 can be referenced as *L***[1]** or ***(***L***+1)**. The standard array element syntax *L***[1]** is easier to remember.

Program **p02** allocates an array of integers, assigns Fibonacci numbers to elements in the array, prints the array, and reclaims storage for the array.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{    const int SIZE=10;
     int* L=new int[SIZE];
     L[0]=1; L[1]=1;
     for (int a=2;a<SIZE;a++) L[a]=L[a-1]+L[a-2];
     cout << endl;
     cout << "Fibonacci Sequence";
     for (int b=0;b<SIZE;b++) {
          if (b%5==0) cout << endl;
          cout << setw(5) << L[b];
   }
   cout << endl;
   delete[] L;
   return 0;
}
```

Figure 4. Program **p02**.

Program **p02** prints:

```
    1    1    2    3    5
    8   13   21   34   55
```

Categories of storage are organized by their lifetime.

1. Storage for constants is allocated during compilation. Constant storage is reclaimed when the program returns control to the operating system.
2. Storage for static data, data declared **static** or data declared outside the scope of a function, is allocated when the program is invoked by the operating system – when function *main* is called. Storage for static data is reclaimed when the program returns control to the operating system.
3. Storage for local data, data declared within the scope of a function, is allocated when the function is called and reclaimed when the function returned. Storage for local data are allocated on the invocation stack.
4. Storage for dynamically allocated data is allocated and reclaimed under the control of the program. However storage for all dynamically allocated data is allocated after function *main* is called and reclaimed when function *main* returns control to the operating system.

An activation record is allocated each time a function is called and reclaimed when the function returns. Since a function must return control to its caller, activation records form a stack. Local variables are allocated in the activation record to which they belong. The diagram in Figure 5 shows the relationship between local and dynamic storage for program **p01**. The diagram in Figure 5 is a snapshot of storage just before the unnamed integer referenced by variable *p* is reclaimed.

In one storage model, activation records form a stack that grows toward memory with larger addresses. Storage for dynamic memory grows toward memory with smaller addresses. It is possible for the activation record stack and dynamic memory to collide under unusual circumstances. The activation record for function *main* has control information in it not pictured in Figure 5. Presumably the top portion of dynamic memory has already been allocated. Note variable p and how it points to an anonymous integer allocated in dynamic storage. The value of the anonymous integer is **5**.

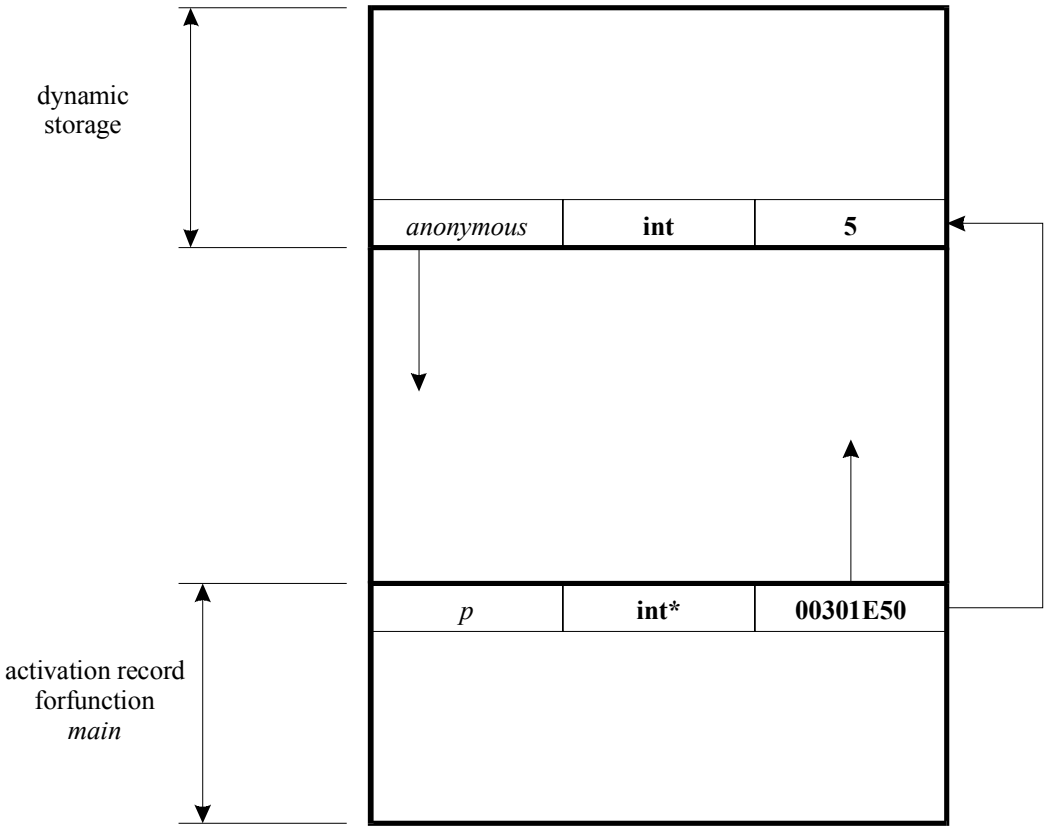

Figure 5. Storage for local and dynamic variables

Exercises:

1. Type in program **p01**, compile it, and make it work according to the description given in this note.
2. Type in program **p02**, compile it, and make it work according to the description given in this note.
3. Write program **p03** that reads a list of real numbers, stores them in a dynamically allocated array whose elements have type **double**, finds the average, and prints every value above the average. The array is reclaimed just before program **p03** returns. The first value read is an integer specifying the number of real numbers in the list.