CS 101:
Computer Programming and Utilization

Jul-Nov 2017

Umesh Bellur
(cs101@cse.iitb.ac.in)

Lecture 16: Representing variable length entities (introducing new and delete)

A programming problem

- Design a scheme to store names of the students in your class
- ”Natural solution”: Use a 2d array of characters, store ith name in ith row.
  - Rowsize will have to be as large as length of longest name.
  - Most rows will be empty. Inefficient use of memory.
- Is there a better scheme?
- Another similar problem: how to store polygons with possibly different number of sides.

Outline

- **Heap memory**
  - Basic primitives for allocation and deallocation
- Issues in managing heap memory
- Detailed Example
  - A class for representing text strings.
  - Use in storing names of students.

What you already know: the activation frame memory

- The main mechanism we have studied for defining variables: variable definitions given in the text of the main program, or of some function.

- Memory for the variables is allocated in the activation frame of the function when control reaches the variable definition statement.

- When control exits the block containing the definition, the memory is freed, or deallocated.

The Heap memory

- In C++ there is a **separate, reserved region of memory** called the **Heap memory**, or just the Heap.

- It is possible to ***explicitly request that memory for a certain variable be allocated in the heap***.

- When there is no more use for the variable thus allocated, the ***program must explicitly return the memory to the heap***. After the memory is returned, it can be used to satisfy other memory allocation requests in the future.

Example: A variable on the heap to store a Book object

```
class Book{
  char title[100];
  double price;
};
Book *bptr;
bptr = new Book();
OR   = new Book;
bptr->price = 399;


…
delete bptr;
```

- new: asks for heap memory
- Must be followed by type name T
- Memory for storing one variable of type T is allocated on the heap.
- new T returns address of allocated memory.
- Now use the memory!
- After the memory is no longer needed, it must be returned by executing delete.
- new and delete are reserved words, also operators.

What happens behind the scenes for `new` and `delete`

- Some bookkeeping goes on behind the scenes to keep track of which part of the heap is currently in use.

- What you are guaranteed: in response to a `new` operation, ***you will get memory that is not currently allocated to another request.***

- The same region of memory can be allocated to two requests, but only if the first request releases it (`delete`) before the second request is made.

Allocating arrays on the heap

```
char* cptr;
cptr = new char[10];
// allocates array of length 10.
// array can be accessed as usual
// cptr[0],…,cptr[9]

delete[] cptr;
// When not needed.
// Note: delete[] not delete
```

Storing many names

```
char *names[100];
// array of pointers to char
for(int i=0; i<100; i++){
  char buffer[80];
   cin.getline(buffer);
  int L = 80+1;
  // string length. +1 for '\0'.
  names[i] = new char[L];
  // copy buffer into names[i];
   strcpy(names[i], buffer);
}
```

- The jth character of the ith name can be accessed by writing `names[i][j]` as you might expect.

Remarks

- Allocation and deallocation is simple and convenient.

- However, experience shows that managing heap memory is tricky and prone to errors.
  - forgetting to deallocate (`delete`) memory.
  - Referring to memory that has been deallocated. ("Dangling reference")
  - Destroying the only pointer to memory allocated on the heap before it is deallocated ("Memory Leak")

Dangling reference

```
int* iptr;
iptr = new int;
*iptr = …;
delete iptr;
*iptr = ...;    // dangling reference!
```

- In the last statement, iptr points to memory that has been returned to the OS, and so should not be used.
- In particular, it might in general be allocated for some other request.
- Here the error is obvious, but if there are many intervening statements it may not be.

Memory Leak 1

```
int *iptr;
iptr = new int; // statement 1
iptr = new int; // statement 2
```

- Memory is allocated in statement 1, and its address, say A, is stored in `iptr`. However, this address is overwritten in statement 2.
- Memory allocated at address A cannot be used by the program because we have destroyed the address.
- However, we did not return (`delete`) that memory before destroying the address. So the heap allocation functions think that it has been given to us.
- The memory at address A has become useless! "Leaked"

Memory Leak 2

```
{
    int *iptr;
    iptr = new int; // statement 1
}
```

- Memory is allocated in statement 1, and its address, say A, is stored in `iptr`.
- When control exits the block, then `iptr` is destroyed.
- Memory allocated in statement 1 cannot be used by the program because we do not know the address any longer.
- However, we did not return (`delete`) that memory before destroying the address. So the heap allocation functions think that it has been given to us.
- So the memory at address A has become unusable!

Simple strategy for preventing memory leaks

- Suppose a certain pointer variable, `ptr` , is the only variable that contains the address of a variable allocated on the heap.

- We must not store anything else into `ptr` and destroy its contents.

- When `ptr` is about to go out of scope, (control exits the block in which `ptr` is defined) we must execute `delete ptr;`

Simple strategy for preventing dangling references

- Why we get a dangling reference:
- There are two pointers, say `aptr` and `bptr` which point to the same variable on the heap.
- We execute `delete aptr;`
- Later we dereference `bptr`, not realizing the memory it points to has been deallocated.
- ***Simple way to avoid this: Ensure that at all times, each variable on the heap will be pointed to only by one pointer!***
- See the book for more on this.

Summary: Avoiding dangling references and memory leaks

- Ensure each variable allocated on the heap is pointed to by exactly one pointer at any time.
- If `aptr` points to a heap variable, then before executing `aptr = …` execute `delete aptr;`
- If `aptr` points to a heap variable, and if control is about to exit the block in which `aptr` is defined, then execute `delete aptr;`
- We can wrap all this in members. Next.

A class for representing character strings

- We would like to build a `String` class in which we can store character strings of arbitrary length, without worrying about allocating memory, memory leaks, dangling references.
- We should be able to create `Strings`, pass them to functions, concatenate them, search them, and so on.

A program we should be able to write

```
int main(){
  String a, b, c;
  a = "pqr";
  b = a;
  {
   String c = a + b;
   // concatenation
   c.print();
  }
  String d[2];
  d[0] = "xyz";
  d[1] = d[0] + c;
  d[1].print();
}
```

- Our class should **enable** us to write the program shown.
  - Creation of string variables
  - Assignment
  - Concatenation
  - Printing
  - Declaring arrays
- All this requires memory management, but that should happen behind the scenes, without memory leaks, dangling pointers.

Basic ideas in designing `String`

- We will store the string itself on the heap, while maintain a pointer `ptr` to it inside our class.
- The string will always be terminated using the null character '\0'.
- When no string is stored in our class, we will set `ptr` to NULL.
  - NULL (=0) : standard convention, means pointer is invalid.
  - NULL pointer different from NULL character.
- To avoid dangling references and memory leaks, we will ensure that
  - Each `ptr` will point to a distinct char array on the heap.
  - Before we store anything into `ptr`, we will delete the variable it points to.
  - When any `ptr` is about to go out of scope, we will delete it.
- Other designs also possible – later.

The class definition

```
class String{
  char* ptr;
  String(){            // constructor
    ptr = NULL;        // initially empty string
  }
  void print(){        // print function
    if(ptr != NULL)
      cout << ptr;
    else
      cout <<“NULL”;
  }
  // other member functions..
};
```

Assigning a character string constant

- We allowed a character string constant to be stored in a `String`:

```
String a;
a = “pqr”;
```

- Thus, we must define member function `operator=`
  - Character string constant is represented by a `const char*` which points to the first character in the string.
  - So we will define a member function `operator=` taking a `const char*` as an argument.

What should happen for a = “pqr”;

- `a.ptr` must be set to point to a string on the heap holding “pqr”.
- Why not set `a.ptr` to point to “pqr” directly?
  - Member `ptr` must point to the heap memory. The character string constant “pqr” may not be on the heap.
- a.ptr may already be pointing to some variable on the heap.
  - We are guaranteed that no other pointer points to that variable, so we must `delete a.ptr` so that the memory occupied by the variable is returned to the heap.

The code

```
String& operator=(const char* rhs){
  // release the memory that ptr already points to.
  delete ptr;

  // make a copy of rhs on the heap
  // allocate length(rhs) + 1 byte to store '\0'
  // Assume a length function defined in book
  ptr = new char[length(rhs)+1];

  // actually copy.  Function scopy defined in book
  scopy(ptr, rhs);

  // We return a reference to the class to
  // allow chaining of assignments.
  return *this;
}
```

Assigning a String to another String

- We want to allow code such as

```
String a, b;
a = “pqr”;
b = a;
```

- The statement `b = a;` will cause a call `b.operator=(a)` to be made.
- So we need a member function `operator=` which takes a `String` as argument

The code

```
String& operator=(const String &rhs){
    // We must allow self assignment.
    // If a self assignment, do nothing.
     if(this == &rhs) return *this;

     // Call the previous "=" operator.
     *this = rhs.ptr;

    return *this;
}
```

The destructor

- The destructor gets called when a `String` object goes out of scope, i.e. control exits the block in which it is defined.
- Clearly, we must delete `ptr` to prevent memory leaks.

```
~String(){
  delete ptr;
}
```

- Note that this will work even if `ptr` is NULL; in such cases `delete` does nothing.

The copy constructor

- Copy constructor is like an assignment, except that
  - we know that the destination object is also just being created, and hence its `ptr` cannot be pointing to any heap variable.
  - we don't need to return anything.
- Hence this will be a simplified version of the assignment operator:

```
String(const String &rhs){
  ptr = new char[length(rhs.ptr)+1];
  scopy(ptr,rhs.ptr);
}
```

The [] operator

- To access the individual characters of the character string, we define operator[].

```
char& operator[](int i){
  return ptr[i];
}
```

- We are returning a reference, so that we can change characters also, i.e. write something like

```
String a; a = “pqr”;
a[0] = a[1];
```

  – This should cause `a` to become “`qqr`”.

Concatenation: + operator

- We use a+b to mean the concatenation of a , b.

```
String operator+(const String &rhs) {
  String res;  // result
 // Allocate space for the result.
 res.ptr = new char[length(ptr)+length(rhs.ptr)+1];
  // Copy the string in the receiver into the result.
   scopy(res.ptr, ptr);
   // Copy the string in rhs but start at length(ptr)
   // New version of scopy defined in book.
   scopy(res.ptr, rhs.ptr, length(ptr));

  return res;
}
```

Remarks

- We have given the definitions of all the member functions needed to be able to perform assignment, passing and returning from functions, concatenation etc. of `String` objects.
- The code given should be inserted into the definition of String.

How to store many names: using our string class

- Here is a program to read 100 names and store them.

```
int main(){
  String names[100];
  char buffer[80]
  for(int i=0; i<100; i++){
     cin.getline(buffer,80);
     names[i] = buffer;
  }
  // now use the array names[] however you want.
}
```

- If we use our class `String`, we do not need to mention memory allocation, it happens automatically in the member functions.

Concluding Remarks

- The class `String` that we have defined performs memory allocation and deallocation behind the scenes, automatically.
- From the point of the user, `String` variables are similar to or as simple as `int` variables, except that `String` variables can contain character strings of arbitrary length rather than integers.
- C++ Standard Library contains a class `string` (all lowercase) which is a richer version of our `String` class.