Unit 11

Object-Oriented Programming in C++

Storage Management

CONTENTS

Unit 11 - Storage Management

Operators new and delete 11-5
Dynamic Storage Allocation 11-7
Using new and delete 11-9
Hiding Storage Management 11-13
A Better Implementation of class String 11-15
String Operations for: 11-19
  Creation 11-19
  Initialization 11-21
  Initialization with a String 11-23
  Destruction 11-25
  Assignment 11-27
  Concatenation 11-29
Private Functions 11-31

Exercises 11 Ex - Lab Exercises

Answers 11 Ans - Exercise Answers

Objectives

At the end of this unit we will be able to:

- create and destroy objects on the free store
- hide details of memory management in a class

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Operators new and delete

When a variable is declared, it is created and destroyed according to well defined rules: If it is global or static, it is created before the start of `main` and destroyed after the end of `main`. If it is local, it is created upon entry to the block in which it is declared, and destroyed upon exit from that block. Compilers usually need to know the size of global, static, and local objects, so it is usually impossible to create a global, static, or local variable whose size is not known at compile time.

If a function needs to create an object that will still exist after the function returns, or if it needs to create an object whose size is not known at compile time, it can create the object on the free store (or heap), in which case it will last until it is explicitly destroyed.

An object can be created on the free store with the operator **new**. **new** is applied to a type name, and creates an object of that type. **new,** like the `malloc` function of C's standard library[1], yields a pointer to the object it created. That object can be destroyed by applying the operator **delete** to the pointer.

1. **new** and **delete** manage the heap by calling `malloc` and `free`, so that libraries using `malloc` and `free` can be called from C++ programs using **new** and **delete.**

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Operators new and delete

operator **new**

- creates an object on the "free store"
  - — allocates storage
  - — calls constructor
- returns pointer to object, or NULL if out of storage
- object exists until explicitly destroyed

operator **delete**

- destroys an object created with **new**
  - — calls destructor
  - — frees storage

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

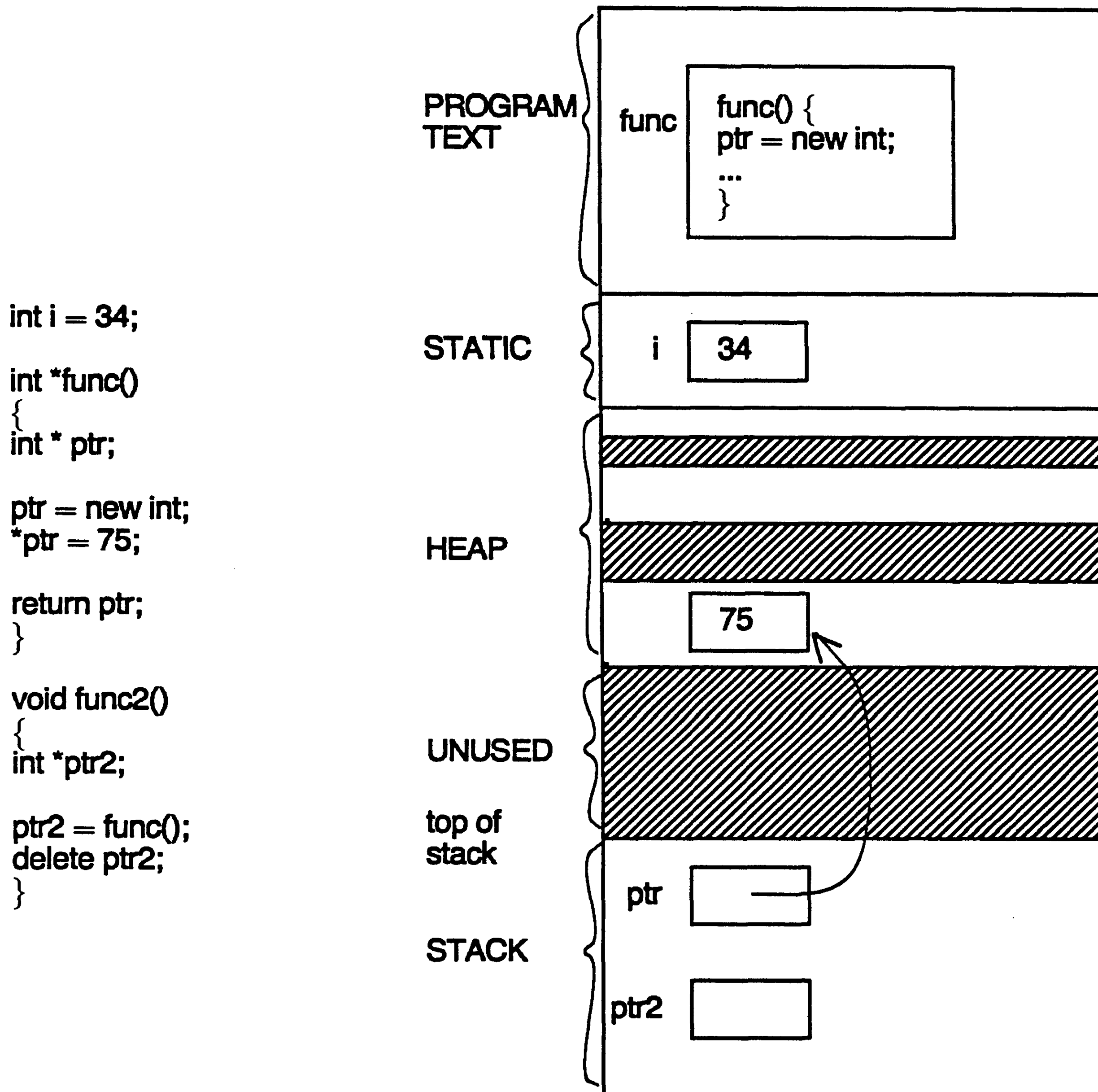
Dynamic Storage Allocation

The size of all static and global variables is known at compile time, so a fixed size piece of memory can be set aside for them when the program starts up. Local variables are always created and destroyed in LIFO (last in, first out) order, so the memory for them can be organized as a stack. The compiler knows nothing about the timing of creation and destruction of the free store, so it is organized as a heap.

The facing page shows a typical division of memory into text, static, stack, and heap memory.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Dynamic Storage Allocation



Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Using new and delete

Objects allocated with **new** can be initialized by specifying the initial value in parenthesis after the type name.

Arrays of objects can be allocated with **new** by placing the array size in square brackets after the type name. In this case, **new** returns a pointer to the first element of the array. There is no way to give initial values for the elements of the array, so the class must have either a constructor with no arguments or no constructors at all. Note that the size of the array need not be known at compile time. The size of the array must be specified when the array is deleted, if the class has a destructor. Since a destructor may be added to a class later, we recommend specifying the array size of all arrays destroyed with **delete.** **new** and **delete** can be used for either built-in or user-defined types.

```
main(int argc, char *argv[])
{
    // Dynamic Allocation
    int *ptr1 = new int;
    if (ptr1 == 0) exit(1);
    *ptr1 = 12;
    printf("%d\n", *ptr1);

    // Dynamic Allocation and Initialization
    int *ptr2 = new int(12);
    if (ptr2 == 0) exit(2);
    printf("%d\n", *ptr2);

    // Dynamic Allocation of an Array
    int *ptr3 = new int[argc];
    if (ptr3 == 0) exit(3);
    ptr3[4] = 12;
    printf("%d\n", ptr3[4]);

     // Destroy Objects
    delete ptr1;
    delete ptr2;
    delete[argc] ptr3;

    return 0;
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Using new and delete

```
#include "String.h"

main(int argc, char *argv[])
{
    // Dynamic Allocation
    String *ptr1;
    ptr1 = new String;
    if (ptr1 == 0) exit(1);
    *ptr1 = "Hello, world";
    (*ptr1 + "\n").print();

    // Dynamic Allocation and Initialization
    String *ptr2;
    ptr2 = new String("Hello, world");
    if (ptr2 == 0) exit(2);
    (*ptr2 + "\n").print();

    // Dynamic Allocation of an Array
    String *ptr3;
    ptr3 = new String[argc];
    if (ptr3 == 0) exit(3);
    ptr3[argc-1] = "Hello, world";
    (ptr3[argc-1] + "\n").print();

    // Destroy Objects
    delete ptr1;
    delete ptr2;
    delete[argc] ptr3;

    return 0;
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Hiding Storage Management

A class can hide details of memory management from users of the class. If our `String` class contains a pointer to a dynamically allocated block of characters (instead of an array of characters), but the member and friend functions handle all the storage allocation, then users of class do not need to know about the pointer or the dynamically allocated memory. They will just continue to use the `String` operations, without having to worry about how those operations are implemented.

Since the size of a dynamically allocated piece of memory does not need to be known until the memory is allocated, the member and friend functions can allocate only as much memory as is needed for the `String`. This avoids the waste of space that used to occur when a 5 byte string was stored in the 129 byte array called `text`, and allows the creation of `Strings` that are longer than 128 bytes.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Hiding Storage Management

A class can hide details of memory management.

- declare a pointer in the private section
- member and friend functions manage pointer & storage
- users of the class don't worry about memory management

class `String` can be enhanced

- variable length `Strings`
- existing code will still work

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

A Better Implementation of class String

The facing page shows the new class String. We have replaced the fixed-size array `text` with the pointer `heap_ptr` (which will point to the memory we allocate from the heap). The new comment gives the new representation invariant: The pointer must point at heap storage, and that heap storage must contain a null-terminated array of characters.

We will leave the declarations of the member and friend functions alone, so that code that uses class `String` will not need to be changed (although it will have to be re-compiled).

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

A Better Implementation of class String

```
class String {
public:
     String();
     String(const char *);
     String(const String &);
     ~String();

     const char *as_char_pointer() const;

     String &operator=(const String &);
     int  length() const;
     int  read();
     void print() const;

     const char &operator [](int) const;
     char & operator [](int);
     String substring(int start, int len) const;

     static int n_concatenations();
     friend String operator+(const String &, const String &);
private:
   // heap_ptr must always point to a
   // dynamically allocated,
   // null terminated
   // array of characters

   char *heap_ptr;
     static int concat_calls;
};
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Memory Allocation

The facing page shows the memory used for string objects with the old and new implementations of class `String`.

```
String s = "hello";
```

With the old implementation, a local variable of type `String` would take up `max_string_length + 1` bytes of stack storage (labeled s.text), even if the string is only 5 bytes long. With the new implementation, the only stack memory used is for the pointer (labeled s.heap_ptr). The member and friend functions must ensure that the pointer points to dynamically allocated memory that holds the text of the `String`. The memory for the text does not have to be any longer than the length of the `String` + 1 (for the null byte). When we replace the old implementation with the new, the users of class `String` will not have to change their programs. The users will only notice a few minor changes in `Strings`: they no longer have a size limit, and the space and time requirements of the `String` operations may have changed. In all other ways, the new `String` objects look just like the old, to the user.

```
String *s_ptr;
s_ptr = new String("Hello");
```

The declaration of s_ptr does not change with our new class `String`. With either implementation, it is a pointer on the stack.

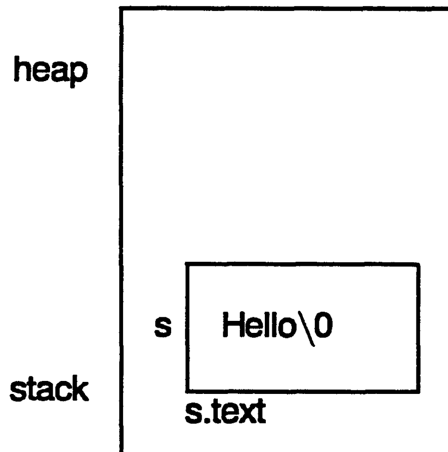
When we created a string with **new,** the old implementation allocated `max_string_length + 1` bytes (for the `text` member) on the heap, even if the string was only 5 bytes long. The new implementation allocates space for the pointer "heap_ptr" on the heap, and points it to another region of heap storage that holds the characters of the string. The users will only notice a change in the length limit, and space and time used by the operations.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

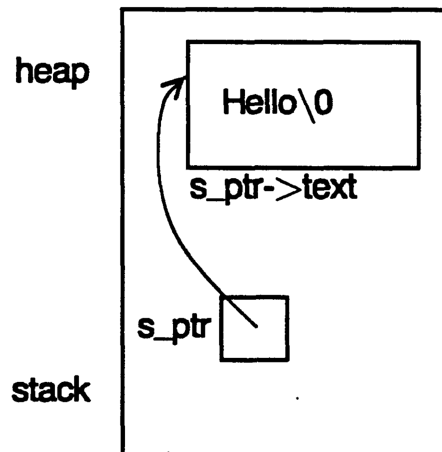
String Memory Allocation

OLD IMPLEMENTATION

String s = "Hello";

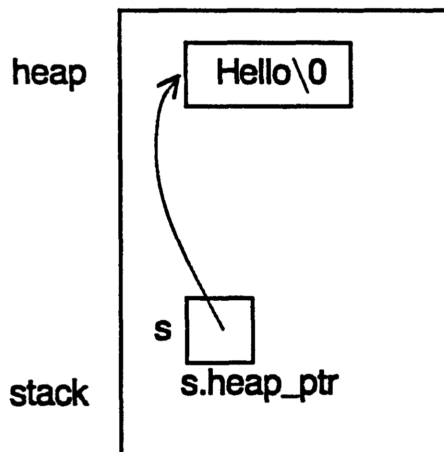


String *s_ptr = new String("Hello");

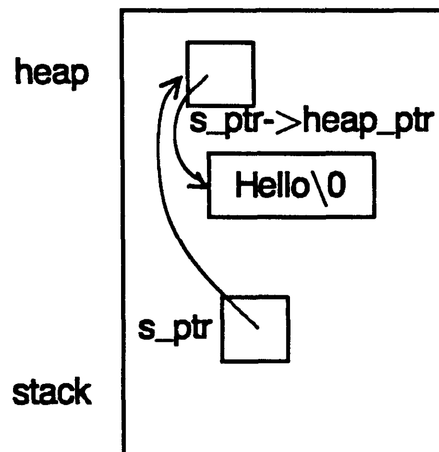


NEW IMPLEMENTATION

String s = "Hello";



String *s_ptr = new String("Hello");



Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Creation

The string constructor's job is to ensure that a newly created String obeys the representation invariant. It therefore allocates a single null byte on the heap, and points heap_ptr to it.

Every String that is created will undergo a two-step process: first, the memory for the data declared in the class is allocated automatically. Then, the constructor does whatever is necessary to finish the creation of the string.

If a String variable is declared, the memory for the data member (the pointer heap_ptr) will come from the stack or the static area. If a String is created with **new,** the memory for the data member will come from the heap. In either case, the constructor will be called to finish the creation.

```
#include "String.h"

String *func1()
{
    String s;  // stack storage for s.heap_ptr,
    // then constructor is called

    String *s_ptr;  // stack storage for s_ptr,
    // no constructor

    s_ptr = new String;   // heap storage for s_ptr->heap_ptr,
    // then constructor is called

    return s_ptr;
}
```

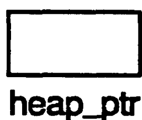
Note that, if there were no String constructor, most of the other String operations would probably cause core dumps when they tried to use the uninitialized pointer heap_ptr. Even the assignment operator needs to use the pointer to free the old storage of the object on the left (as we will see in a few pages), so there could even be a core dump if the user declared a String and immediately assigned it a value.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

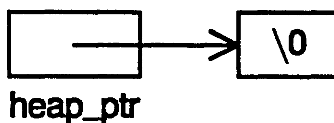
String Creation

```
String s;
// OR
s_ptr = new String;
```

BEFORE CONSTRUCTOR:



AFTER CONSTRUCTOR:




```
String::String()
{
    heap_ptr = new char;
    if (heap_ptr == (char *) NULL) {
      fprintf(stderr,
          "Insufficient storage for empty string");
      exit(1);
    }
    *heap_ptr = '\0';
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Initialization

The constructor to initialize strings with `char *` values are is shown on the facing page. If we did not provide this constructor, users could not initialize `Strings` with `char *` values.

This constructor will also be used to cast `char *` values to type `String`.

```
#include "String.h"

String *func4(String s)  // stack storage for s.heap_ptr,
                         // then constructor is called
{
    String local = "Hello world";  // stack storage for local.heap_ptr,
                                   // then constructor is called

    String *s_ptr;  // stack storage for s_ptr,
                    // no constructor

    s_ptr = new String("Hello world");
                                   // heap storage for s_ptr->heap_ptr,
                                   // then constructor is called

    return s_ptr;
}

void test_func4()
{
    String *ptr = func4("test");
    delete ptr;
}
```

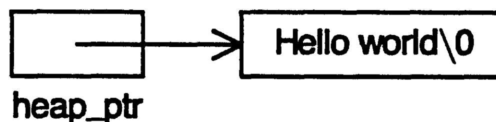
Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Initialization

```
String s = "Hello world";
// OR
s_ptr = new String("Hello world");
```

BEFORE CONSTRUCTOR:

AFTER CONSTRUCTOR:




```
String::String(const char *init)
{
    heap_ptr = new char[strlen(init)+1];
    if (heap_ptr == (char *) NULL) {
      fprintf(stderr,
          "Insufficient storage for string \"%s\"\n", init);
      exit(1);
    }
    strcpy(heap_ptr, init);
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Copying

If a class does not provide a constructor for initializing with a value of its type (e.g., initializing a String with a String), C++ will use the mechanism compatible with structure copying in C: it will simply copy the memory for the members of the initializing String into the memory of the String being created. With our original class String, this mechanism would have worked.

The new class String, however, must control this form of initialization, because the default mechanism would just copy the heap_ptr of the initializing String over the heap_ptr of the string being created. That would result in two pointers to the text of the original String - not a satisfactory situation, because changes to one would change the other. Our String::String(const String &) constructor allocates space for a copy of the characters in the original String, and copies the characters.

This constructor will also be used to copy String type arguments.

```
#include "String.h"

String *func5(String s)  // stack storage for s.heap_ptr,
                         // then constructor is called
{
    String local = s;  // stack storage for local.heap_ptr,
                       // then constructor is called

    String *s_ptr;  // stack storage for s_ptr,
                    // no constructor

    s_ptr = new String(local); // heap storage for s_ptr->heap_ptr,
                               // then constructor is called

    return s_ptr;
}

void test_func5()
{
    String argument = "test";
    String *ptr = func5(argument);
    delete ptr;
}
```

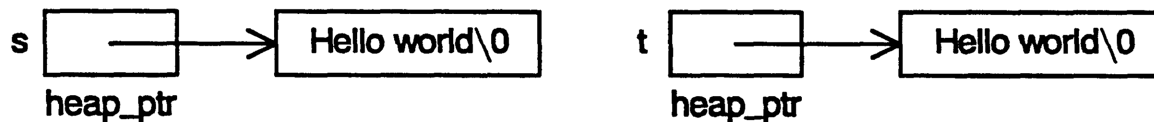
Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Copying

```
String s = t;
// OR
s_ptr = new String(t);
```

BEFORE CONSTRUCTOR:

AFTER CONSTRUCTOR:




```
String::String(const String &init)
{
    heap_ptr = new char[strlen(init.heap_ptr)+1];
    if (heap_ptr == (char *) NULL) {
      fprintf(stderr,
          "Insufficient storage for string \"%s\"\n",
          init.heap_ptr);
      exit(1);
    }
    strcpy(heap_ptr, init.heap_ptr);
}
```


Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Destruction

Our original class `String` did not require a destructor, since the default mechanism for destruction (freeing the storage for the data members declared in the class) would destroy the `String`. Now, if we had no destructor, the memory for the `heap_ptr` would be freed, but the heap memory containing the characters would not. Therefore, our destructor must free the storage pointed to by `heap_ptr`.

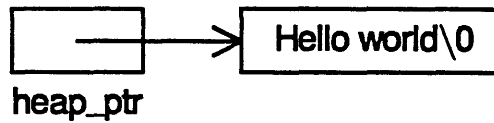
```
#include "String.h"

void func2()
{
    String s = "Hello world";
    String *s_ptr;
    s_ptr = new String("Hello world");

    delete s_ptr;   // destructor is called,
                   // then storage for s_ptr->heap_ptr
                   // is returned to the heap

}     // upon return from func, destructor is called for s
        // and storage for s.heap_ptr is returned to the stack
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Destruction

```
delete s_ptr;
// OR s destroyed automatically
```

BEFORE DESTRUCTOR:



AFTER DESTRUCTOR:

```
String::~String()
{
    delete heap_ptr;
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Assignment

Our original class String did not need to control assignment of strings, since the default mechanism for assignment (assigning the data members of the object on the right to the members of the object on the left) would work. The new class String, however, must control String assignment, because the default mechanism would just copy the heap_ptr of the String on the right over the heap_ptr of the String on the left. That would result in two pointers to the characters of the right hand String, with the characters of the left hand String still allocated, with no pointers to them (so there is no way to free this memory later). Therefore, our operator= frees the storage for the characters of the String on the left (which is the invoking object of the operator= member function), allocates enough storage to hold a copy of the right hand String's characters, and copies them.

```
#include "String.h"

void func3()
{
    String s = "Hello world";
    String t;

    t = s;    // operator=(const String &)
}
```

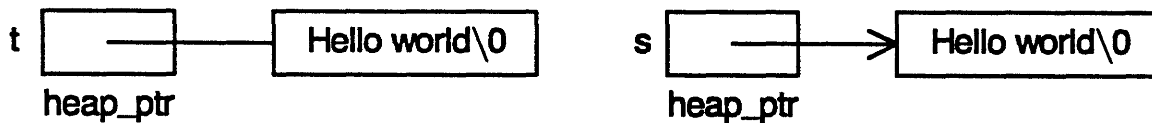
Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Assignment

```
t = s;
```

BEFORE ASSIGNMENT:

AFTER ASSIGNMENT:




```
String &String::operator=(const String &rhs)
{
    if (this != &rhs) {      // in case of a = a
      delete heap_ptr;    // get rid of old text

      heap_ptr = new char[rhs.length()+1];
      if (heap_ptr == (char *) NULL) {
           fprintf(stderr,
                "Insufficient storage for string \"%s\"\n",
                rhs.heap_ptr);
           exit(1);
      }
      strcpy(heap_ptr, rhs.heap_ptr);
    }

    return *this;
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Concatenation

The concatenation function must also do some memory allocation. The temporary it declares to hold the result will start out as a null `String` (its `heap_ptr` pointing to a single null byte). The `operator+` must free the single null byte, and allocate enough storage to hold the result of the concatenation.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String Concatenation

```
#include "String.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

String operator+(const String &lhs, const String &rhs)
{
    String both;

    delete both.heap_ptr;
    both.heap_ptr = new char[ lhs.length()+rhs.length()+1 ];
    if (both.heap_ptr == (char *) NULL) {
      fprintf(stderr,
          "Insufficient storage for \"%s\" + \"%s\"\n",
          lhs.heap_ptr, rhs.heap_ptr);
      exit(1);
    }

    String::concat_calls++;

    strcpy(both.heap_ptr, lhs.heap_ptr);
    strcat(both.heap_ptr, rhs.heap_ptr);

    return both;
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Private Functions

Several string member functions allocate text for a string and fill it with some data (the constructors and the assignment operation). If we put the common code in a private member function, that member function can be called by the constructors and assignment operations, but not by the users of class String.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Private Functions

Member functions in the private section:

- Can be called by other members & friends
- Can not be called from any other functions
- Are useful when several member functions have similar code

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String::alloc_and_set

The String member function alloc_and_set allocates heap storage for a string, and fills that storage with the characters passed to it. The constructors and operator= can be written easily in terms of alloc_and_set.

```
String::String(const String &s)
{
    alloc_and_set(s.heap_ptr);
}

String::String()
{
    alloc_and_set("");
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

String::alloc_and_set

```
class String {
// same public section
private:
    void alloc_and_set(const char *);
    char *heap_ptr;
};
void String::alloc_and_set(const char *s)
{
    heap_ptr = new char[strlen(s)+1];
    if (heap_ptr == (char *) NULL) {
      fprintf(stderr,
          "Insufficient storage for string \"%s\"\n", s);
      exit(1);
    }
    strcpy(heap_ptr, s);
}

String::String(const char *s)
{
    alloc_and_set(s);
}
String &String::operator=(const String &rhs)
{
    if (this != &rhs) {
      delete heap_ptr;
      alloc_and_set(rhs.heap_ptr);
    }

    return *this;
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Summary

Once we have decided on the right set of defining operations for a class, we can turn our attention toward making that class efficient and flexible.

The original class `String` served an important role: it let us focus our attention on getting the right set of operations without being distracted by implementation details (and later we could focus our attention on implementation details without being distracted by questions of which operations we really wanted). It also let users of class `String` start their coding while class `String` was under development. Users could start with the old class `String`, and later switch to the new class `String` without having to re-write their code.

When the implementation of a class involves storage management, constructors and destructors become even more important. They can be used to ensure that memory is allocated when an object is created, and to ensure the memory is properly freed when the object is destroyed. A destructor, assignment operator, and copy constructor that were all optional in the original class `String` are necessary to make the new class `String` work.

There is still room for improvement in the implementation of our class `String`. For most applications, a keeping a length field is more efficient than relying on a null byte at the end of an array. The `String` class could be re-written to keep a length field instead of a null-terminated array. The copy constructor and the assignment operator always copy one `String's` characters into another `String`. This is not really necessary if neither string is ever changed. We would re-write class string to keep a count of the number of strings sharing the characters and copy these characters only when necessary. Either of these changes could be made without forcing users of class string to re-write their code.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Summary

The details of memory management for a class may be hidden by that class's member functions and friends:

- constructor functions can handle storage allocation when objects are created
- a destructor function can free storage when objects are destroyed
- a copy constructor can handle storage allocation when objects are copied (for function calls and returns)
- the assignment operator can handle storage allocation during assignment
- other defining operations (such as +, substring, etc) may also need to allocate storage.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Lab Exercises

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

UNIT 11

Lab Exercises

1. In this lab, you will be asked to create an entire class. This will give you a chance to combine C++ features from many parts of this course, and test your ability to decide when each language feature is needed. You will not need to use any files from the previous labs.

   Change to the *unit11/int_array* directory. Create a class **Int_array**, representing an array of integers, according to the following descriptions. The class declaration should be in the file *Int_array.h*. You should put the code for implementing the functions in the file *int_array.c*. A test program has been provided for each step of this problem, in the files *test1a.c*, *test1b.c*, *test1c.c*, *test1d.c*, and *test1e.c*. Also, a *Makefile* has been provided for your convenience.

   a. The size of an **Int_array** must be given when the **Int_array** is created:

   ```
   Int_array ia(10);
   ```

   b. Elements in the **Int_array** can be accessed with the **[ ]** operator. Note that this exercise does *not* require an **operator=** function.

   ```
   ia[0] = 100;
   ia[1] = 10;
   printf("first 2 elements should be 100 and 10: %d, %d.\n", ia[0], ia[1]);
   ```

   c. If an illegal index is given, the program halts:

   ```
   ia[9] = 17;
   ia[5000] = 508; // program stops
   ```

   d. **Int_arrays** can be assigned; if they are, the **Int_array** on the left takes on the size and all the elements of the **Int_array** on the right. Note that this exercise *does* require an **operator=** function.

   ```
   Int_array ia2(7);

   ia2[0] = -1;
   ia2 = ia;
   printf("ia2[9] should be 17: %d\n", ia2[9]);
   ```

   e. If an **Int_array** is passed by value to a function, the function receives a copy of the **Int_array**:

```
test(ia2);
printf("ia2[9] should still be 17: %d\n", ia2[9]);

// where test is the following:

void test(Int_array copy)
{
    printf("copy[9] should be 17: %d\n", copy[9]);
    copy[9] = -1;
    printf("copy[9] changed to -1: %d\n", copy[9]);
}
```

You can compile and execute each test program by entering '**make probl*n***' (*n* = a, b, c, d, or e); you can compile and execute all of them by entering '**make**'; or, you can compile and execute them directly using the commands:

```
$CC test1a.c int_array.c -o test1a
$test1a
$CC test1b.c int_array.c -o test1b
$test1b
$CC test1c.c int_array.c -o test1c
$test1c
$CC test1d.c int_array.c -o test1d
$test1d
$CC test1e.c int_array.c -o test1e
$test1e
```

SUMMARY	
DIRECTORY	unit11/int_array
DECLARATION	Int_array.h (new)
IMPLEMENTATION	int_array.c (new)
TEST PROGRAM	test1*x*.c (x = a, b, c, d, e)

UNIT 11

Lab Exercises (Answers)

1. The completed class **Int_array** is shown after the discussion of which parts are necessary for each of the lab exercises.

   a. If we are to provide the size of an **Int_array** when we create it, we must consider that size as the initial value of the array, and create a constructor (**Int_array::Int_array(int size)**) to handle that form of initialization. Since we do not wish to allow un- initialized **Int_arrays**, we will not provide a default constructor.

      Most of the code for this constructor is in the function **allocate**, which allocates storage for the integers, points the array pointer to that storage, and sets both **legal_size** and **actual_size**. The code in **allocate** was written as a separate function because it will be needed in several other member functions of class **Int_array**. This was not apparent at first, and the class was written once without **allocate**, and then revised to include this private function.

      The two size fields will be discussed when they become important later (in answer 1d). Note that after the constructor **Int_array::Int_array(int size)** has finished, all 3 statements in the comment labeled "Rep invariant" in the header file *Int_array.h* must be true.

   b. See part (c).

   c. The two **operator[]** functions allow indexing into **const** and non-**const** **Int_arrays**. Both check to determine if the requested index is within the size of the array, so that a program will halt if it tries to index an illegal element of the array.

   d. The function **Int_array::operator=(const Int_array &rhs)** will handle the assignment of one **Int_array** to another. We must write this function because the built-in mechanism for assignment will not work properly for **Int_arrays**. The built-in mechanism would assign all the members (the sizes and the pointer) of one **Int_array** to the other. If this happened, both **Int_array**'s pointers would point to the same piece of memory, and changes to one **Int_array**'s elements would affect the other **Int_array**.

      The original code for the assignment operator simply freed the old storage, allocated enough to hold the new contents of the array, and copied the elements (as is done in class **String** in the lecture notes).

```
Int_array &Int_array::operator=(const Int_array &rhs)
{
   int i;

   delete array;
   allocate(rhs._size);

   for (i = 0; i < _size; i++)
      array[i] = rhs.array[i];

   return *this;
}
```

Note that there is no need to free the old array and allocate a new one if the existing array is large enough. If the existing array is large enough, we can just change the **legal_size** of the **Int_array** without freeing up and re-allocating the storage. If we choose to do this, we should save the **actual_size** of the allocated array in a separate field, so that we can check it later if a new value is assigned to the **Int_array**. That is what is done by the class **Int_array** shown below.

e. The constructor **Int_array::Int_array(const Int_array &init)** will be used to initialize **Int_arrays** with values of type **Int_array**. This happens when we initialize a local variable, or when an **Int_array** is passed to a function with an **Int_array** parameter. The actual argument given in a function call is used as the initial value for the parameter declared in the function.

   This constructor allocates enough space, and copies the elements of the initializing value into the object being created.

FILE: **Int_array.h**

```
class Int_array {
public:
   Int_array(int size);
   Int_array(const Int_array &init);
   ~Int_array();

   int &operator[](int index);
   const int &operator[](int index) const;

   Int_array &operator=(const Int_array &rhs);

private:
   void allocate(int size);

   int *array;
   int legal_size;
   int actual_size;

   // Rep invariant:
   // legal_size gives the number of legal indexes in the array
```

```
    // array points to a dynamically allocated array of actual_size ints
    // actual_size >= legal_size
};
```

FILE: int_array.c

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

void Int_array::allocate(int size)
{
     array = new int[size];
     if (array == NULL) {
         fprintf(stderr,
                "Insufficient storage for Int_array of size %d.\n",
                size);
         exit(1);
     }
     actual_size = legal_size = size;
}

Int_array::Int_array(int size)
{
     allocate(size);
}

Int_array::~Int_array()
{
     delete array;
     // delete[actual_size] not needed for array of built-in type
}

Int_array::Int_array(const Int_array &init)
{
     int i;
     allocate(init.legal_size);
     for (i = 0; i < legal_size; i++)
         array[i] = init.array[i];
}

int &Int_array::operator[](int index)
{
     if (index < 0 || index >= legal_size) {
         fprintf(stderr, "illegal index\n");
         exit(1);
     }
     return array[index];
}

const int &Int_array::operator[](int index) const
{
     if (index < 0 || index >= legal_size) {
         fprintf(stderr, "illegal index\n");
         exit(1);
     }
     return array[index];
```

```
}

Int_array &Int_array::operator=(const Int_array &rhs)
{
     if (rhs.legal_size > actual_size) {   // allocate more storage
         delete array;
         allocate(rhs.legal_size);
     }
     else { // re-use existing storage if rhs is same size or smaller
         legal_size = rhs.legal_size;
     }

     int i;
     for (i = 0; i < legal_size; i++)
         array[i] = rhs.array[i];

     return *this;
}
```