CS319: Scientific Computing (with C++)

Introduction to classes

Week 4: 9am and 4pm, 01 February 2017

Encapsulation

Idea: bind program data to the program code (i.e., functions) that manipulate it.

The next bit is really important: not just to C++, but for writing robust scientific computing code.

Within an object, code and data may be either

- **Private**: accessible only to another part of that object, or
- **Public**: other parts of the program can access it even though it belongs to a particular object. The public parts of an object provide an **interface** to the object for other parts of the program.

An object should be thought of as an **Abstract Data Type** (ADT): a specialised type of variable that the user can define.

There are many important examples of "built-in" C++ classes, such as `string`, `cin` and `cout`. But we'll leave those until later, and first study how to make our own.

Example: a stack

In C++, *encapsulation* is implemented using the `class` keyword. The example we'll consider is a **stack** – a *LIFO* (Last In First Out) queue.

There is already a C++ implementation of a `stack`. It is part of the **Standard Template Library (STL)**. *We reinvent the wheel here only because it is a nice example that includes most of the key concepts associated with classes in C++. We will study the STL later in CS319.*

Our stack will permit two primary operations:

- an item may be added to the top of the stack: `push()` ;
- an item may be removed from the top of the stack: `pop()`.

These then are our interfaces to the stack. Hence these will be **public**.

Example: a stack

For the stack itself, the following must be maintained:

- an array containing the items in the contents;
- a counter/index to the top of the stack.

These are *private* to the class.

We choose this example because it is obvious that

- `push()` and `pop()` are the interfaces to the object–they are declared as *public*;
- the contents of the stack, and the counter of the number of objects in it, need only be visible to the object itself; hence they are private.

In our example there is also a public function to initialise the stack.

class

The basic syntax for defining a class:

```
class class-name {
private:
    ...     // private functions and variables
public:
    ...     // public functions and variables
};
```

class-name becomes a new object type—one can now declare objects to be of type *class-name*.

This is only a declaration. Therefore,

- functions are not defined, though the prototype is given,
- variables are declared but are not initialised,
- the declaration block is delineated by { and }, and terminated with a semicolon.

class

As mentioned our class has two `private` members

- `contents`: a *`char`* array of length `MAX_STACK` the array containing the stacked items.
- `top`: an *`int`* that stores the number of items on the stack.

It hasd three `public` member functions:

(a) `init()` sets the stack counter to 0. It has no arguments or return value.

(b) `push()` adds an item to the stack. The character to be added is its only argument.

(c) `pop()` takes no argument but returns the removed item.

```
class stack {
private:
  char contents[MAX_STACK];
  int top;
public:
  void init(void );
  void push(char c);
  char pop(void );
};
```

To define the functions associated with a particular class we use

1. the name of the class, followed by
2. the *scope resolution operator* `::` , followed by
3. the name of the function.

We now define the three (public) functions: `init()`, `push()` and `pop()`.

The `init()` is required only to set the value of `top` to zero:

```
void stack::init(void)
{
  top=0;
}
```

Note that we didn't have to declare the (private) variable `top`.

The push() function takes as its only argument a single character. It adds the character to the stack and increments the index to the top of the stack.

```
void stack::push(char c)
{
  contents[top]=c;
  top++;
}
```

The pop() function doesn't take any arguments (void). It removes the item from the stack by returning the top entry and decrementing top.

```
char stack::pop(void)
{
  top--;
  return(contents[top]);
}
```

The first item in the stack is at position 0, the second is a position 1, the 3rd is at position 2, etc. So when top=n then there are n items in the stack but the top one is actually located in contents[n-1].

Now that our class `stack` has been declared, and its functions defined, we can declare objects to be of type `stack`, e.g.,

```
stack s1, s2;
```

We can refer to the functions `s1.pop()` and `s2.push(c)`, say, because these are public members of the class. We cannot refer to `s1.top` as this variable is private to the class and is hidden from the rest of the program.

To use the objects, we might have a `main()` function that behaves as follows:

- Declare and initialise a stack object `s1`;
- Push the characters `'C','S','3','1','9'` onto the stack;
- The stack's contents are removed and output to the console using `cout`.

01Stack.cpp ← link!

```
40 int main(void )
   {
42   stack s1;

44   s1.init();

46   s1.push('C');
     s1.push('S');
48   s1.push('3');
     s1.push('1');
50   s1.push('9');

52   std::cout << "Popping ... " << std::endl;

54   std::cout << s1.pop() << std::endl;
     std::cout << s1.pop() << std::endl;
56   std::cout << s1.pop() << std::endl;
     std::cout << s1.pop() << std::endl;
58   std::cout << s1.pop() << std::endl;

60   return (0);
   }
```

Constructors

Suppose we wanted to change the `stack` class so that the user can choose the maximum number of elements on the stack...

In the example above, the function `init()` is used explicitly to initialise the variable `top`. However, there is an initialisation mechanism called a **Constructor** that is built into the concept of a class.

CONSTRUCTOR

A **Constructor** is a public member function of a class

- that shares the same name as the class, and
- is executed whenever a new instance of that class is created.

Constructors

This function may contain such code as you would find in any function. However, is good practice to only use A constructor for initialization.
For example, suppose we change the declaration of the `stack` class as show here:

```
class stack {
public:
  stack(void); // Constructor. No return type
  void push(char c);
  char pop(void );
private:
  char contents[MAX_STACK];
  int top;
};
```

We then replace the `init()` function with:

```
stack::stack(void )
{
  top=0;
}
```

Note that the constructor as no explicit return type.

Now whenever an objects of type `stack` is created, e.g., with `stack s1`, the functions `s1.stack()` is called automatically – and *`s1.top`* is set to zero.

Pointers

The next topic we'll study is **Dynamic Memory Allocation**.
But first we need to get our heads around the topic of **Pointers**.

Take notes:

02Pointers.cpp ← link!

```
12    char a='W', b='Q';
      char *where;

      std::cout << "The variable \"a\" stores " << a << std::endl;
16    std::cout << "The variable \"b\" stores " << b << std::endl;
      std::cout << "The variable \"a\" is stored at the address " <<
18      (void *)&a << std::endl;
      std::cout << "The variable \"b\" is stored at the address " <<
20      (void *)&b << std::endl;

22    where = &a;
      std::cout << "The variable \"where\" stores "
24              << (void *) where << std::endl;
      std::cout << "... and that in turn stores " <<  *where << std::endl;
```

Dynamic Memory Allocation

Our stack example from earlier is quite limited in many ways. One of then is that the stacks can only store at most `MAX_STACK` items.

It would be useful if

- we could have stacks of different sizes, and
- the user/programmer could choose the size.

To add this functionality, we will use two new (to us) C++ operators for dynamic memory allocation and deallocation: `new` and `delete`. (There are also functions `malloc()`, `calloc()` and `free()` inherited from C).

The `new` operator is used in C++ to allocate memory. The basic form is

```
var = new type
```

where *type* is the specifier of the object for which you want to allocate memory and *var* is a pointer to that type.

If insufficient memory is available then `new` will return a NULL pointer or generate an exception.

To use `new` allocate space for the integer *top* and initialise it to zero:

```
top = new int(0);
```

To dynamically allocate an array:

- First declare a pointer of the right type:

  ```
  char *contents;
  ```

- Then use `new`

  ```
  contents = new char[MAX_STACK];
  ```

When it is no longer needed, the operator `delete` releases the memory allocated to an object.

The basic syntax is

```
delete var;
```

where *var* is a pointer previously allocated with `new`.

To "delete" an array we use a slightly different syntax:

```
delete [] array;
```

where *array* is a pointer to an array allocated with `new`.

We now make the following modifications to the `stack` implementation (for full implementation, see `03StackConstructor.cpp`)

```
class stack {
public:
  stack(void);
  void push(char c);
  char pop(void );
private:
  char *contents;
  int top;
};
```

Here we have changed `contents` so that it is a pointer.

```
stack::stack(void)
{
  contents = new char [MAX_STACK];
  top=0;
}
```

Destructors

Complementing the idea of a constructor is a **destructor**. This function is called

- for a local object – whenever it goes out of scope,
- for a global object – when the program ends.

The name of the destructor is the same as the class, but preceded by a tilde:

```
class stack {
private:
  char *contents;
  int top;
public:
  stack(void );
  ~stack(void );
  void push(char c);
  char pop();
};
```

```
stack::~stack()
{
  delete [] contents;
}
```

The Constructor again...

The example we had earlier of a constructor was particularly basic, not least because is its parameter list is void. More commonly, one passes arguments to the constructor that can be used, e.g.,

- to set the value of a data member;
- dynamically size an array using `new`.

However, one should still provide a default constructor (i.e., one with no arguments).[1]

```
class stack
{
private:
  char *contents;
  int top;
public:
  stack(void);
  stack(unsigned int StackSize);
  void push(char c);
  char pop(void );
};
```

```
stack::stack(void)
{
  top=0;
  contents = new char[MAX_STACK];
}

stack::stack(unsigned int StackSize)
{
  top=0;
  contents = new char[StackSize];
}
```

[1] This is for illustration. Better again: use one constructor, but with a default argument value.

I/O means "Input/Output. So far, we have taken input from the keyboard, typically using cin, and sent output to a terminal window, using cout.

These are examples of **streams**: flows of data to or from your program. Moreover, they are examples of **objects** in C++.

In this section, we'll study how to manipulate these streams in C++, including writing to and reading from files.

But first, some more information about cout and cin.

The objects cout and cin are objects and are manipulated by their **methods**, i.e., public member functions and operators.

Methods:

- width(int x) – minimum number of characters for next output,
- fill(char x) – character used to fill with in the case that the width needs to be elongated to fill the minimum.
- precision(int x) – sets the number of significant digits for floating-point numbers.

Code – width

```
for (int i=65; i<123; i++)
{
  std::cout.width(8);
  std::cout << i;
  std::cout.width(3);
  std::cout << (char) i;
  if ( (i%5) == 4)
    std::cout << std::endl;
}
```

Output

```
 65  A      66  B      67  C ...
 70  F      71  G      72  H ...
 75  K      76  L      77  M ...
 80  P      81  Q      82  R ...
 85  U      86  V      87  W
 90  Z      91  [      92  \
 95  _      96  `      97  a
100  d     101  e     102  f
105  i     106  j     107  k
110  n     111  o     112  p
115  s     116  t     117  u
120  x     121  y     122  z
```

Code – width, fill

```
std::cout.fill('0');
for (int i=0; i<8; i++)
{
  std::cout.width(6);
  std::cout << rand()%200000 <<std::endl;
}
```

Output

```
089383
130886
092777
036915
147793
038335
085386
```

Code – precision

```
double Pi=3.1415926535;
for (int i=1; i<=10; i++)
{
  std::cout.precision(i);
  std::cout << "Pi (correct to "<< i << " digits) is "
            << Pi << std::endl;
}
```

Output

```
Pi (correct to 1 digits) is 3
Pi (correct to 2 digits) is 3.1
Pi (correct to 3 digits) is 3.14
Pi (correct to 4 digits) is 3.142
Pi (correct to 5 digits) is 3.1416
Pi (correct to 6 digits) is 3.14159
Pi (correct to 7 digits) is 3.141593
Pi (correct to 8 digits) is 3.1415927
Pi (correct to 9 digits) is 3.14159265
Pi (correct to 10 digits) is 3.141592654
```

- `setw` – like `width`
- `left` – Left justifies output in field width. Used after `setw(`*n*`)`.
- `right` – right justify.
- `endl` – inserts a newline into the stream and calls flush.
- `flush` – forces an output stream to write any buffered characters
- `dec` – changes the output format of number to be in decimal format
- `oct` – octal format
- `hex` – hexadecimal format
- `showpoint` – show the decimal point and some zeros with whole numbers

Others: setprecision(n), fixed, scientific, boolalpha, noboolalpha, ...

Need to include `iomanip`

All of the C++ programs we have looked at so far took their input from the *standard input stream*: this was usually the keyboard.

Example:

```
std::cout << "Enter an inteter: ";
std::cin >> i;
```

Although, for example, the *standard input stream* can be redirected to a file, it is usually necessary to open a file **from within the program** and take the data from there.

The same is true for writing to a file.

To do either of these takes in C++ we create a **file stream** and use it just as we would `cin` or `cout`.