CS319: Scientific Computing (with C++)

Week 4: Introduction to classes

9am 02 March, and 4pm, 03 March, 2021

1. Part 1: Functions - default arguments
2. Part 2: Binary and bitwise operators
3. Part 3: Recursion
4. Part 4: Encapsulation
5. Part 5: `class`
   - Example – a stack
   - `class`
6. Part 6: Constructors
7. Part 7: Dynamic memory allocation
   - Pointers
   - DMA
   - `new`
   - `delete`
8. Part 8: Destructors
   - The Constructor again...

New class times

recorded.

	Mon	Tue	Wed	Thu	Fri
9 – 10		LECTURE	✗		
10 – 11		LAB			
11 – 12					
12 – 1					
1 – 2		LAB			
2 – 3					
3 – 4					
4 – 5			LECTURE		

recorded

1. The recorded class on Wednesdays at 9.00 moves to **Tuesday at 9.00**.
2. The recorded class on Thursdays at 16.00 stays.
3. **New lab times: Tuesday 10.00-10:50, and 13.00-13.50**. You should try to attend at least one of these.
4. Little, if any, of the “lab” times will be recorded.
5. This may all change again towards the end of the semester.

Part 1: Functions - default arguments

CS319 – Week ~~3~~ 4
Week 4: Introduction to classes

Start of ...

PART 1: Functions - default argument values

Part 1: Functions - default arguments

In C++, one can also define functions that have assigned default values:

```
int mult(int a, int b=1, int c=1) // from 00Mult.cpp
{
   return(a * b * c);
}
```

default arguments.

This means that, if the user fails to provide the second and third arguments to the function, it is assumed that they are both 1.

Example

```
std::cout << "mult(1) = " << mult(1);
std::cout << "mult(1,2) = " << mult(1,2);
std::cout << "mult(1,2,3) = " << mult(1,2,3);
```

(Take b & c as 1)

⇐ c defaults to 1

↑

a=1, b=2, c=3

Part 1: Functions - default arguments

CS319 – Week 3
Week 4: Introduction to classes

END OF PART 1

Part 2: Binary and bitwise operators

CS319 – Week ~~3~~ 4
Week 4: Introduction to classes

Start of ...

PART 2: Binary numbers and bitwise operators

Part 2: Binary and bitwise operators

Last week, we say that all data in C++ (and all other languages, etc) is really stored in binary.

To help us get a better understanding of binary numbers, and operations on them, we'll now study how to convert numbers from decimal (Base 10) to binary (Base 2).

This will also motivate an example of programming a recursive function in C++.

First, recall that a **decimal (i.e., base 10) integer** is made up of the digits 0, 1, 2, ... 9, and that the k^{th} digit (from the right) is the coefficient of 10^{k-1}.

$2\ 0\ 3\ 4 = \quad 4 + 30 + 0 + 2000$

4×10^0

3×10^1

0×10^2

2×10^3

$10^0 = 1$

$10^1 = 10$

$10^2 = 100$

$10^3 = 1{,}000$

Part 2: Binary and bitwise operators

Next, recall that a **binary (base 2) integer** is made up of the "bits" 0 and 1, and that the k^{th} digit (from the right) is the coefficient of 2^{k-1}.

Example: Here's how to convert from binary to decimal.

$1 \quad 1 \quad 1 \quad 1 = 8 + 4 + 2 + 1 = 15.$

$+$

$1 \times 2^0 = 1$

$1 \times 2^1 = 2$

$1 \times 2^2 = 4$

$1 \times 2^3 = 8$

Eg

$1\,0\,0\,1 = 8 + 0 + 0 + 1 = 9.$

$1 \times 2^0 = 1$

$0 \times 2^1 = 0$

$0 \times 2^2 = 0$

$1 \times 2^3 = 8$

Part 2: Binary and bitwise operators

There are several important operations on binary numbers, that don't really have decimal equivalents, including

Bit-wise AND:

15 "AND" 9:

$$\begin{array}{rcccc} & 1 & 1 & 1 & 1 \\ \text{AND} & 1 & 0 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 \end{array}$$

$= 9.$

Bit-wise OR:

Bit-wise EXCLUSIVE OR (XOR):

"0 = false, 1 = true".

These are implemented in C++ using &, |, and ^, respectively.

"Bit-wise" – means apply the operator to every pair of bits, in order

Part 2: Binary and bitwise operators

There are several important operations on binary numbers, that don't really have decimal equivalents, including

~~Bit-wise AND:~~

Bit-wise OR: → 15 or 9 :

$$\begin{array}{rcccc} & 1 & 1 & 1 & 1 \\ \text{or} & 1 & 0 & 0 & 1 \\ \hline & 1 & 1 & 1 & 1 \end{array}$$

= 15

Bit-wi~~se EXCLUSIVE OR (XOR):~~

or

These are implemented in C++ using &, |, and ^, respectively.

Truth table

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Part 2: Binary and bitwise operators

There are several important operations on binary numbers, that don't really have decimal equivalents, including

Bit-wise AND:

Bit-wise OR:

Bit-wise EXCLUSIVE OR (XOR): 15 "xor 9" =6

$$\begin{array}{r} 1111 \\ \text{xor } 1001 \\ \hline 0110. \end{array}$$

$= 6$

These are implemented in C++ using &, |, and ^, respectively.

The way these work is they take ints (so, base 10) as inputs, convert them to binary, apply the operator, and return result in Base 10.

Part 2: Binary and bitwise operators

To check how these operators work, we'll need to be able to convert from ~~binary to~~ decimal: to binary.

a string is a seq of chars.

01Binary.cpp

```
std::string Int_to_Binary(int a)
{
   std::string A="";
   for (int i=(int)log2(a); i>=0; i--)
   {
      if ( a >= pow(2,i))
      {
         A=A+"1";
         a=a-pow(2,i);
      }
      else
         A=A+"0";
   }
   return(A);
}
```

number of bits in the number.

Eg: if $a = 15$, then $\log_2(a) = 3.8\ldots$?

And $(int) \log_2(a) = \underline{\underline{3}}$ (Rounds down).

We'll return to a recursion-based implementation later...

Part 2: Binary and bitwise operators

Next, the calling part (modified from the actual code to simplify formatting):

01Binary.cpp (main function)

```
int a, b, c;

std::cout << "Input two integers: ";
std::cin >> a >> b;
std::cout << "You entered: " << a << " and " << b;

std::cout << a << " = " << Int_to_Binary(a) << std::endl;
std::cout << b << " = " << Int_to_Binary(b) << std::endl;

c = a^b;
std::cout << "XOR: a^b = " << c << " = " << Int_to_Binary(c);
c = a&b;
std::cout << "AND: a&b = " << c << " = " << Int_to_Binary(c);
c = a|b;
std::cout << " OR: a|b = " << c << " = " << Int_to_Binary(c);
```

Part 2: Binary and bitwise operators

CS319 – Week ~~3~~ 4
Week 4: Introduction to classes

END OF PART 2

Part 3: Recursion

CS319 – Week ~~3~~4
Week 4: Introduction to classes

Start of ...

PART 3: Recursion

A function that calls itself (very like mathematical induction).

Part 3: Recursion See Pascals triangle.

Many problems in scientific computing can be solved by replacing the problem by a similar but simpler one, and solving that instead.

Here are a few very simplistic examples:

- Suppose we want to compute $x = a^b$, where b is a positive integer. We could first compute a^{b-1}, and then set $x = (a)(a^{b-1})$. The process can be repeated:

so, say $x = pow(a,b) = a * pow(a, b-1)$
$= a * a * pow(a, b-2) = \dots = a * a * a \times \cdots \times pow(a,1)$

- Suppose we want to compute $x = n!$, where n is a positive integer. We could first compute $(n-1)!$, and then compute $x = (n)(n-1)!$.

Both these are candidates for computation by recursion.

$$n! = n \underbrace{(n-1)(n-2)(n-3)\cdots(2)\cdot(1)}_{(n-1)!}$$

Part 3: Recursion

02Power.cpp

```
float Power(float a, unsigned int b);  // compute a to power of b
int main()
{
  float a, c;
  int b;

  std::cout << "Input float, a, and nonnegative integer, b: ";
  std::cin >> a >> b;
  std::cout <<  "You entered: a=" << a << " and b=" << b;

  c = Power(a,b);
  std::cout << a << " to the power of "<< b << " is " << c;
  return(0);
}

float Power(float a, unsigned int b)
{
  if (b==0)
    return(1);
  else
    return( a*Power(a, b-1));
}
```

Power is recursive.

Part 3: Recursion

As mentioned above, we can write a recursive decimal-to-binary converter. Here it is below. *Can you work out how it works?*

```
// A simple example of a recursive algorithm:
// converting from decimal to binary
// Based on Shapira "Solving PDEs in C++", Section 1.18
#include <iostream>
#include <math.h>
int Binary(int a);   // return the binary representation of a
int main(void)
{
...
}

int Binary(int a)
{
  if (a<=1)
    return(a);
  else
    return(10*Binary(a/2) + a%2);
}
```

Part 3: Recursion

CS319 – Week 3
Week 4: Introduction to classes

END OF PART 3

Part 4: Encapsulation

CS319 – Week ~~3~~ 4
Week 4: Introduction to classes

Start of ...

PART 4: Encapsulation

Part 4: Encapsulation

Encapsulation

Idea: create a single entity in a program that combines data with the program code (i.e., functions) that manipulate that data.
In C++, a description/definition of such entities is called a **class**, and an instance of such an entity is called an **object**.

That is, like a variable is a single instance for a `float` (for example), then an object is a single instance of a `class`.

A class 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`, and objects, such as `cin` and `cout`. But we’ll leave those until later, and first study how to make our own.

Part 4: Encapsulation

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.

It is referred to a **"data hiding"**, an important concept in software design.

Part 4: Encapsulation

CS319 – Week ~~3~~ 4
Week 4: Introduction to classes

END OF PART 4

Finished here at 9.50, Tuesday.

Part 5: class

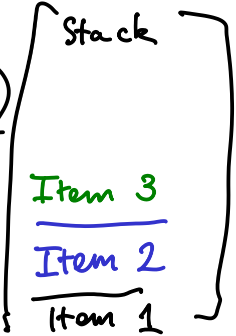
Start here Wed @ 4pm.

CS319 – Week ~~3~~ 4

Week 4: Introduction to classes

Start of ...

PART 5: class



Part 5: `class`

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.*

. . . .

The name of our class will be `MyStack`. It 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**.

Part 5: `class` — 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.

Part 5: class

The basic syntax for defining a class:

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

Here class private public are keywords in C++

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.

Part 5: class

Defined elsewhere

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 has three public member functions:

(a) init() sets the stack counter to 0. No arguments or return value.

(b) push() adds an item to the stack. One argument: the character to be added.

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

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

private

public.

Part 5: class — function definitions

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

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

Recall std::cout

↑

Part 5: class — function definitions

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 MyStack::push(char c) {
  contents[top]=c;
  top++; → increase top by 1.
}
```

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 MyStack::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].

Part 5: `class` main()

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

```
MyStack 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 could have a `main()` function that behaves as follows:

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

Part 5: class main()

03MyStack.cpp

```
int main(void ) {
  MyStack s;

  s.init();

  s.push(’C’);
  s.push(’S’);
  s.push(’3’);
  s.push(’1’);
  s.push(’9’);

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

  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;

  return (0);
```

CS319 – Week 3
Week 4: Introduction to classes

END OF PART 5

Part 6: Constructors

CS319 – Week ~~3~~ 4
Week 4: Introduction to classes

Start of ...

PART 6: Constructors

(for class).

Part 6: Constructors

Suppose we wanted to change the MyStack 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.

(ie run automatically)

Part 6: Constructors

Constructors may contain any code you like; but it is good practice to only use them for initialization.

As an example, we'll change the declaration of the `stack` class as shown here:

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

// Version 2

← instead of the init() function,

Eg 10

Part 6: Constructors

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

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

function name .

Note that the constructor as no explicit return type.

Now whenever an objects of type `MyStack` is created, e.g., with

  `MyStack s`,

the functions `s.MyStack()` is called automatically – and *s.top* is set to zero.

class name

Part 6: Constructors

CS319 – Week ~~3~~ 4
Week 4: Introduction to classes

END OF PART 6

Part 7: Dynamic memory allocation

CS319 – Week ~~3~~ 4
Week 4: Introduction to classes

Start of ...

PART 7: Dynamic memory allocation

Part 7: Dynamic memory allocation

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:

If we define an integer x, e.g.,

```
int x=5;
```

Then x is associated with a memory address, and the value 5 is stored there.

We can access that address as &x.

Similarly, we can define special variables, called POINTERS, to store memory addresses.

Eg `int *p;` Then `p=&x;` is OK.

Part 7: Dynamic memory allocation

Pointers

04Pointers.cpp

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

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

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

Part 7: Dynamic memory allocation

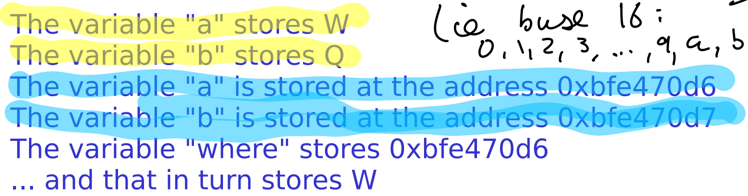
Our stack example from earlier is quite limited in many ways. One of then is that the stacks can only store at most `MAX_STACK` items.

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).

0x – means hex
(ie base 16:
0, 1, 2, 3, ..., 9, a, b, c, d, e, f)



```
The variable "a" stores W
The variable "b" stores Q
The variable "a" is stored at the address 0xbfe470d6
The variable "b" is stored at the address 0xbfe470d7
The variable "where" stores 0xbfe470d6
... and that in turn stores W
```

Part 7: Dynamic memory allocation

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. (Eg int, float)

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];
```

stores the address of first element of the array.

Part 7: Dynamic memory allocation delete

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.

Part 7: Dynamic memory allocation

We now make the following modifications to the stack implementation (for full implementation, see 05MyStackConstructor.cpp)

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

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

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

CS319 – Week 3
Week 4: Introduction to classes

END OF PART 7

Finished here Wed @ 4.50.

Part 8: Destructors

CS319 – Week 3
Week 4: Introduction to classes

Start of ...

PART 8: Destructors

Part 8: 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 MyStack {
private:
  char *contents;
  int top;
public:
  MyStack(void );
  ~MyStack(void );
  void push(char c);
  char pop();
};
```

```
MyStack::~MyStack()
{
  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), or one with a default argument list.

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

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

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