C++ intro

C++ intro and Differences in C-like programming in C++

- C++ intro
- Basic I/O
- Call by reference using &
- Memory allocation - new and delete
- Const

The class

- The class
- Class scope
- Member types (public, private, friend)
- Functions in C++: overloading, default variables, const functions and inline functions
- Constructors and destructors
- Creating objects
- When constructors and destructors are called
- Copy constructors

Differences between C-like programming and C++ programming

- C++ recognizes a C code; every legal statement in C is also legal in C++.
  C++ includes C.
- C is action oriented, so the building blocks are functions. C++ is object oriented, so the building blocks are classes.
- There are different ways to accomplish the same goal in C and in C++.

Basic I/O

- Use the **cin** and **cout** objects to perform the basic I/O manipulations. The header iostream.h is required.
- The I/O operators are the *extraction operator* << and the *insertion operator* >> . They can also be cascaded.
- The endl argument is used to flush the buffer.

cout can be used to perform a printf() task; it is a formatted output – you don't have to mention the variable type (character, string, integer or float) in order to plot it:
```
cout<<"print text, integers "<<12<<" and floats "<<11.72<<endl;
```
output:
```
print text, integers 12 and floats 11.72
```

cin is used to get input. It is also formatted, and it ignores white spaces:
```
int x;
double y;
cin>>x>>y;
cout<<"received the integer "<<x<<" and the double "<<y<<endl;
```
output:
```
received the integer 5 and the double 2.56
```

You can also use the good old scanf, printf functions to receive input from the user and output them to the standard output.
Example:
```
char charArr[10];
memset(charArr, 0, sizeof(char)*10); //cleanup
scanf("%s", charArr); //receives a string formatted input from stdin into charArr
printf(charArr); // print charArr into stdout
```

Call by reference using & operator

A call by reference can be used without sending a pointer, but rather sending the object's address without the pointer mediator:

```
#include <iostream.h>
void square( int &num ) { num=num*num; }

int main() {
   int x=4;
   cout<<”x before squaring: “<<x;
   square( x );
   cout<<”\tafter squaring: “<<x<<endl;
   return 0;
}
```

output:

```
x before squaring: 4      after squaring 16
```

Memory allocation

- The function **new()** replaced malloc(). No size is needed, only the object's type, and no casting is needed. This way, less programmer-originated errors occour.
- The function **delete()** replaced free(). Deleting an array has to be specified explicitly.
- Do not combine new()-delete() and malloc()-free() directives in the same program!
- **Important:** do not free() memory allocated with new() and do not delete() memory allocated with malloc()!!! This leads to fatal errors.

```
#include <iostream.h>

int main() {

   int *xPtr, *yPtr, *array;

   xPtr = new int(3);
   yPtr = new int;
   *yPtr = 5;

   array = new int[2];
   array[0]=array[1]=7;

   cout<<"xPtr points to "<<*xPtr<<".  xPtr value is "<<xPtr<<endl;
   cout<<"yPtr points to "<<*yPtr<<".  yPtr value is "<<yPtr<<endl;
   cout<<"array elements are "<<array[0]<<" and "<<array[1]<<endl;

   delete xPtr;
   delete yPtr;
   delete[] array;

   return 0;
}
```

output:

```
xPtr points to 3.  xPtr value is 0x007708B0
yPtr points to 5.  yPtr value is 0x007708E0
array elements are 7 and 7
```

Unsuccessful new() operation will return a NULL value.
Use assert() to break out of a program when this happens; the assert.h is required.

```
ptr = new int[10];
assert( ptr !=0 );
```

The class

- The building block of the C++ program.
- Contains member functions and member variables.
- Enables encapsulation.
- Provide different levels of access to members; a class member is private by default.

Class scope and structure:

```
class <name> {
public:
        <public member functions and variables>
private:
        <private member functions and variables>
protected:
        < protected member functions and variables>
};
```

example of class declaration:

```
class Circle {
public:
   Circle();
   void setRadius( int );
   int getRadius();
   double getArea();
private:
   int radius;
};
```

function definition:

```
<return type> <class name> :: <function name>( <function parameters> ) {
   <function body>
}
```

```
Circle::Circle() {  setRadius(0); }
void Circle::setRadius( int r ) { radius = r;   }
int Circle::getRadius() { return radius;   }
double Circle::getArea() {  return 3.14*radius*radius;  }
```

- note the use of the **class scope operator ::** which links the function to the class specifically.
- all member functions recognize all of the classes other members (functions and data) of all types (public, private and protected).

Member types

Public – can be accessed from anywhere outside or inside the object.
Private – only the object's functions or friend functions of the class have access privilege.
Protected – like private, but derived classes also have access permission.

More about C++ functions

Default values

A parameter can be given a default value. It is usually mentioned in the function definition.

```
void numbers( int a=0, int b=0 ) {
   x=a;
   y=b;
}
```

Function call	x	y
numbers(1, 2);	1	2
numbers(5);	5	0
numbers();	0	0

Only the last parameters can be given default values:

```
void numbers( int a=0, int c, int b=0 ) {//ILEAGAL !!!
   x=a;
   y=b;
   z=c;
}
```

The const identifier

const object is a read-only object. Only a const function has access to a const object.
A const function can also access non-const objects. Anyway, it cannot alter member variables.

```
void Circle::printRadius() const {
   cout<<”radius is “<<radius<<endl;
   radius++;     //ILEAGAL!!!
}
```

this use is only allowed in member functions.

a const identifier can identify a function parameter, as done in a C program:

```
void printMessage( const char* str ) {
   cout<<”Message is “<<str<<endl;
   str=0;     //ILEAGAL!!!

}
```

inline functions

member functions that are explicitly defined with the keyword “inline”, or member functions that are defined and implemented inside a class declaration:

```
class MyClass {
   int i;
public:
   void raise() { i = i + 10; }
   void lower();
};
inline void Myclass::lower() { i = i - 20; }
```

- inline functions behave like macros – a literal copy of the compiled function replaces the function call. Thus, inline functions make the code longer, but the “call” is quicker.
- Unlike macro, an inline is a request – the compiler doesn’t have to grant the inline directive.
- Another difference is the type-checking and compile errors – inline functions are compiled!!!
- The use of inline keyword is scarce; inline functions should be very short, so they are often found inside the class’s declaration block.

Function overloading

- You can define many functions with the same name, providing that the return type remains the same.
- Overloaded function can provide related tasks.

```
void printNumber( int num ) { cout<<”print number as integer: “<<num<<endl; }
void printNumber(char *str){ cout<<”print number as string:“<<atoi(str)<<endl; }

printNumber( 1 );
printNumber( “7” );
```

output:

```
print number as integer: 1
print number as string: 7
```

in case no exact function signature is found, the compiler matches and casts the closet call:

```
printNumber( ‘A’ );
```

output:

```
print number as integer: 65
```

which is the integer value of the character ‘A’

sometimes the compiler cannot decide. This can be avoided by calling the overloaded function properly:

an example of not-so-smart calls:

```
int sum( int a, char b ) { return a+(int)b; }
int sum( char a, int b ) { return (int)a+b; }

sum( 12, 89 );
```

Friend functions

- A class can declare “friend” a function that is declared elsewhere, using the friend keyword.
- Friendship is granted, not taken!!!
- A friend function has access to all class members – both private protected and public.
- An entire class can be declared as friend; all members of this friend class have access to all members of the first class.
- A function can be a friend of more than one class.

```
#include <iostream>
class myclass {
  int a, b;   //private by default!!!
public:
  myclass(int i, int j) { a=i; b=j; }
  friend int sum(myclass x);
};

// Note: sum() is not a member function of any class.
// Because sum() is a friend of myclass, it can directly access a and b.
int sum(myclass x) {   return x.a + x.b;   }

int main(){
  myclass n(3, 4);
  cout << sum(n);
  return 0;
}
```

constructors and destructors

- constructors have no return value, not even void; it may, however, receive parameters.
- A constructor name must be the same as its class's name.
- Constructors cannot be declared as static or const
- Otherwise constructors behave the same as regular functions. They can be overloaded, inlined, or have default parameter values.

- destructors are used to accomplish tasks that are vital prior to deletion of an object, mainly freeing memory.
- destructors have no return value and they receive no parameters.
- A destructor name is ~<class name>
- It is possible to define destructors as virtual functions.

```
class Timer {
public:
   Timer() { min=sec=0; }                        //overloaded constructor
   Timer( int s, int m=0 ) { sec=s; min=m; }     //overloaded constructor
   ~Timer() { cout<<"timer destroyed"<<endl; }  //destructor
   int getSec() {return min*60 + sec; }
private:
   int min, sec;
};
```

Creating objects:
An object can in be created in two ways:
Directly, by using an identifier: `Timer t( 20, 4 );`
Indirectly, by using pointer: `Timer *tPtr = new Timer( 20, 4 );`

Accessing public members of directly created objects is done via the . operator:
```
int x = t.getSec();
```
Accessing public members via pointers is done using the -> operator:
```
int y = tPtr->getSec();
```

pointer and reference rules still apply:
```
tPtr = &t;
int x = (&t)->getSec();
int y = (*tPtr).getSec();
```

when constructors and destructors are called

constructors are called in the following cases:
- local objects - when the object is defined in the code explicitly, either by using pointer or directly. Static objects are created before non-static objects, and prior to everything else in the function.
- Global objects – prior to the main() function.

Destructors are called in the following cases:
- when an object is explicitly deleted.
- When an object goes out of scope, implicitly.

Objects are always destroyed in reverse order of their creation.

Static members:

Static members of a class are common to all the objects which are instances of the class.
Example:

```
class Circle {
public:
   Circle(){circlesCounter++;};
   ~Circle(){circlesCounter--;};
   void showCounter();
private:
   int radius;
   static int circlesCounter;
};

int Circle:: circlesCounter =0;

void Circle::showCounter()
{
   cout<<”Number of created circles is: “<< circlesCounter<<endl;
}

int main()
{
   Circle circle1;
   circle1.showCounter();
   Circle* circle2;
   circle2=new Circle;
   cout<<”Circle2: “;
   circle2->showCounter();
   cout<<”Circle1: “;
   circle1.showCounter();

   delete circle2;
   cout<<”After delete: ”;
   circle1.showCounter();
   return 0;
}
```

output:

```
Number of created circles is:1
Circle2: Number of created circles is: 2
Circle1: Number of created circles is: 2
After delete: Number of created circles is: 1
```

Copy constructors

There are two main problems associating with objects and functions: passing an object to function by value, and returning objects from functions.
Both actions require use of copy constructors, and if none is defined the system uses a default copy constructor. The default copy constructor copies objects byte-by-byte and thus can generate numerous problems.

For example, if the original object contains a string, i.e. a char*, the pointer is copied, and the result is that both objects point to the same string, and not to different copies of the same string.
When one object frees this memory, the other remains pointing to an undefined location.

An error generated by returning an object:

```
#include <iostream.h>
#include <string.h>

class sample {
  char *s;
public:
  sample() { s = 0; }
  ~sample() { if(s) delete [] s; cout << "Freeing s\n"; }
  void show() { cout << s << "\n"; }
  void set(char *str);
};

void sample::set(char *str) {     // Load a string.
  s = new char[strlen(str)+1];
  if(!s) {
    cout << "Allocation error.\n";
    exit(1); // exit program if out of memory
  }
  strcpy(s, str);
}

sample input() { // Return an object of type sample.
  char instr[80];
  sample str;

  cout << "Enter a string: ";
  cin >> instr;
  str.set(instr);
  return str;
}

int main() {
  sample ob;
  ob = input();//assign returned object to ob - This causes an error!!!
  ob.show();
  return 0;
}
```

output:

```
Enter a string: Hello
Freeing s
Freeing s
Hello
Freeing s
```

Error ! Trying to delete null pointer.

Demonstrate a problem when passing objects:

```
#include <iostream.h>
#include <stdlib.h>

class myclass {
  int *p;
public:
  myclass(int i);
  ~myclass();
  int getval() { return *p; }
};

myclass::myclass(int i) {
  cout << "Allocating p\n";
  p = new int;
  if(!p) {
    cout << "Allocation failure.\n";
    exit(1); // exit program if out of memory
  }
  *p = i;
}

myclass::~myclass() {
  cout << "Freeing p\n";
  delete p;
}

// This will cause a problem.
void display(myclass ob)  {    cout << ob.getval() << '\n'; }

int main() {
  myclass a(10);
  display(a);
  return 0;
}
```

output:

```
Allocating p
10
Freeing p
Freeing p
```

Error ! Trying to delete null pointer.

Demonstrate a problem when performing: object1=object2

```
#include <iostream>
#include <string.h>

class MyString {
   char *str;
   int size;
public:
   MyString (){};
   MyString (const char *s);
   void set(const char *str);
   void show();
};

MyString::MyString (const char *s){
   str=new char[strlen(s)+1];
   strcpy(str,s);
}
void MyString::set(const char *str){
   if(strlen(s)<size)
   {
      strcpy(str,s);
   }
}
void MyString::show(){
   cout<<str<<’\n’;
}


main()
{
   MyString s1(“first”);
   cout<<”s1: “<<s1.show();
   MyString s2(“second”);
   s1=s2;
   cout<<”s1: “<<s1.show();
   s1.set(“test”);
   cout<<”s2: “<<s2.show();
   return 0;
}
```

output:

```
s1: first
s1: second
s2: test
```

The solution: define a copy constructor!

There are 3 cases when object are copied, and copy constructor is called:

- Passing parameter to a function.
- Returning parameter from a function
- Initialization : Circle c1=c2;

<class name> (const <class name>&<copy name>) { <body>   }

An example:

```
#include <iostream.h>
#include <string.h>

class MyString {
public:
   MyString( const char* );
   MyString( const MyString & );     //copy constructor
   ~MyString() { cout<<"destructing "<<str<<endl;  delete str;}
   char* getString() { return str; }
private:
   char* str;
};

MyString::MyString( const char* s ) {
   str = new char[strlen(s)+1];
   strcpy( str, s );
   cout<<"normal constructor for "<<str<<endl;
}
MyString::MyString( const MyString &other ) {
   str = new char[strlen(other.str)+1];
   strcpy( str, other.str );
   cout<<"copy constructor for "<<str<<endl;
}


void printString( MyString s ) { cout<<"printing "<<s.getString()<<endl;}

int main() {

   MyString s1( "first" );
   MyString s2=s1;
   MyString *s3 = new MyString( "third" );
   printString( *s3 );
   delete s3;
   return 0;
}
```

output:

```
normal constructor for first
copy constructor for first
normal constructor for third
copy constructor for third
printing third
destructing third
destructing third
destructing first
destructing first
```