Intro to C++

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` is required.
- The I/O operators are:
  - the extraction operator <<
  - the insertion operator >>
  - They can also be cascaded.
- The endl argument is used to flush the buffer.

cin: standard input (type of istream)
cout: standard output (type of ostream)
cerr: standard error (type of ostream)

cout

- 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

- 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
```

Reading an unknown number of inputs


```
#include<iostream>
int main() {
  int sum=0, value;
  while (std::cin >> value)
    sum += value;                  //sum = sum + value
  std::cout << "Sum is:" << sum << std::endl;
  return 0;
}
```

- iostream becomes invalid when hitting the end of a file to simulate end of file: Ctrl-d on Unix, Ctrl-z on Windows
- endl: flushing the stream (alternative: std::cout.flush ())

Namespaces

- Namespaces allow to group entities like classes, objects and functions under a name.
- The keyword **`using`** is used to introduce a name from a namespace into the current declarative region. For example:

```
#include <iostream>
using namespace std;
namespace first { int x = 5; int y = 10; }
namespace second { double x = 3.1416; double y = 2.7183; }
int main () {
using first::x;
using second::y;
cout << x << endl;
cout << y << endl;
cout << first::y << endl; cout << second::x << endl;
return 0;
}
```

```
Output:
5
2.7183
10
3.1416
```

namespace std

- Possible use of std as name space :

```
#include<iostream>
using  namespace std;
int main() {
  int sum=0, value;
  while (cin >> value)
    sum += value;                  //sum = sum + value
  cout << "Sum is:" << sum << endl;
  return 0;
}
```

size_t

- **Unsigned integral type**
- size_t corresponds to the integral data type returned by the language operator sizeof and is defined in the <cstring> header file (among others) as an unsigned integral type.

Initialization of arrays

The follow function ill block of memory:

`void * memset ( void * ptr, int value, size_t num );`

Sets the first *num* bytes of the block of memory pointed by *ptr* to the specified *value* (interpreted as an unsigned char).

```
/* memset example */
#include <iostream>
int main () {
  char str[] = "almost every programmer should know memset!";
  memset (str,'-',6);
  cout << str;
  return 0;
}
```

Output :

```
------ every programmer should know memset!
```

Alternative I/O

- 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
cin >> charArr;     // receives a string formatted input
                    // from stdin into charArr
cout << charArr;    // print charArr into stdout
```

Variables declaration

In C++ is possible to declare new variables in all site of the program code.

Example:

```
for ( int i=0, double  num=1.0 ;  i < NUM ; i++){
     loop body ….
}
```

Dynamic memory allocation

- The function new() replaces malloc(). No size is needed, only the objects type, and no casting is needed. This way, less programmer-originated errors occur.
- The function delete () replaces free (). Deleting an array has to be specified explicitly using delete () .
- Do not combine new()–delete() and malloc()–free() directives in the same program!. In
- C++ programming in this course we use only new() and delete ()
- Important: do not free () memory allocated with new() and do not delete () memory allocated with malloc()!!! This leads to fatal errors.

Example


```
#include <iostream>
int main ( ) {
   int *xPtr , *yPtr , *array ; // Pointers
   xPt r = new  int ( 3 ) ; // Dynamic memory  allocation  and assignment
   yPt r = new  int ;
   *yPt r = 5 ;
   array = new  int [ 2 ] ;
   array [0]= array [ 1 ]=7 ;
   cout<<”xPtr points to”<<xPtr<<”  xPtr value is ”<<*xPtr<<e ndl ;
   cout<<” yPtr points to ”<<yPtr<<” . yPtr value is ”<<*yPtr<<endl ;
   cout<<” array elements are ”<<array[0]<<” and ”<<array[1]<<endl ;
 // Delete dynamic memory
   delete xPtr ;   delete yPtr ;
   delete[ ] array ;
   return 0 ;
} //Pay attention --remember to  delete dynamic memory!!!
```

Output

```
xPtr points to 003c38c0. xPtr value is 3
yPtr points to 00363008. yPtr value is 5
Array elements are 7 and 7
```

Dynamic memory allocation

- When we use **new** we must use **delete**

```
int *pi = new int ;
delete pi ;
```

- When we use new[] we must use delete[]

```
int *pi = new int[2] ;
delete[] pi ;
```

- ***Thumb rule:***

The number of times we call new (or new[])

we must call delete (or delete[])

Dynamic memory allocation

//How to allocate a two-dimensional array?

```
int **mat = new int*[N] ;
for ( int i = 0 ; i < N ; ++i){
    mat[i] = new int[N] ;
}
```

// How to delete this memory allocation ?

```
for ( int i = 0 ; i < N ; ++i){
    delete [] mat[i] ;
}
 delete[] mat;
```

Dynamic memory allocation

- Suppose that we know the first dimension of an array at compile time, but the second is variable (so-called jagged arrays)

```
int *mat[5];
for ( int i = 0 ; i < 5; ++i) {
    int N;
    std::cin >> N ;
    mat[i] = new int[N] ;
}
// How to delete this memory allocation ?
for ( int i = 0 ; i < 5 ; ++i){
    delete [] mat[i] ;
}
```

Dynamic memory allocation

What if the inverse is true – we need each cell to contain exactly 5 integers, but we don't know how many cells we'll need?

```
int N ;
std::cin >> N ;
typedef int Cell[5];

Cell *mat = new Cell[N];
// How to delete this memory allocation ?
delete[] mat;
```

Note the power of *typedef* – writing this without defining *Cell* is hard, and the resulting code is unreadable…

const Pointer, const Array

Example:

```
const int num ; // ERROR  (compilation)
const int x = 5 ;
const int *pNum = & x ;
*pNum = 4 ; // ERROR  (compilation)
int *pInt = pNum ;   //ERROR: const int* --> int*
int *otherPNum = (int*)pNum ;   //casting away the const –
                                        //not a good
  practice
const char digits[] = {‘0’, ‘a’, ‘b’ , ‘c’};
digit[3] = ‘u’ ; // ERROR (compilation)
```

const and pointers

Declaration	Description
const int * ptr1;	Defines a pointer to a constant integer: the value pointed to cannot be changed.
int * const ptr2;	Defines a constant pointer to an integer: the integer can be changed, but ptr2 cannot point to anything else.
const int * const ptr3;	Defines a constant pointer to a constant integer: neither the value pointed to nor the pointer itself can be changed.

Local variable

- The scope (or simply the "{ }" braces) of an identifier is the part of the program over which the identifier can be seen (used) by other identifiers and used by other identifiers.
- Local variable is a variable defined inside a function.
  Local variable can be accessed only from the scope where it was defined and cleared automatically at the end of the scope.
- Example (*with compiler errors*):

```
void average(){
  for ( int i =0 ; i < 5 ; i++ ){
    int x , sum = 0 ;
    std::cout<<"Enter the "<< i+1 <<"'th integer:" << std::endl;
     std::cin>>x ;
     sum += x ;
   }
   std::cout<<" The average: "<< sum / i <<std::endl;
}
```

Strings

string


```
#include <iostream>
#include <string>
using  namespace std;
void main() {
   string stringOne("Hello World");   //init
   string stringTwo(stringOne),stringThree;
   int i;
    stringTwo.append(" Again.");     // equivalent to +=
    cout << stringOne << endl;
    cout << stringTwo << endl;
    cout << stringOne + " " + stringTwo << endl;
    cout << "Enter your name: ";
    cin >> stringThree;
    cout << "Hello " << stringThree << endl;
}
```

Output:

```
Hello world
Hello world Again.
Hello world Hello world Again.
```

string (cont.)


```
string k, m, n, o, p;
m = "David";
n = "and";
o = "Goliath";
k = "Solomon";
cout<< k << endl;
cout<< m <<" "<< n <<" "<< o << endl;
p = m + " " + n + " " + o;
cout << p << endl;
p.append(" meet at last!");
cout << p << endl;
m = "Beware the ides of March";
n = m.substr(11,4);
o = m.substr(19);
p = m + n + o;
cout << p <<endl;
cout << p.find("hides") << endl;
```

Returns string::npos if substring isn't found

Classes

Classes in C++ program

- Classes are the main building block of the C++ program.
- Classes contains member functions (methods) and member variables.
- Classes enables encapsulation(we shall see in next lectures).
- Classes provide different levels of access to members.
- A class member is private by default.

Classes structure


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

Classes members definition

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

Function (method) implementation

<return type> <ClassName> : :
<FunctionName>(<function parameters>) {
<function body>
}

All member functions recognize all of the classe other members (functions and data) of all types (public, private and protected).

Function (method) properties

- In general, C++ function follows the same conventions as C functions with several exceptions and additions:
  - Functions can be defined within classes (methods) or outside a class (C type functions)
  - Functions can be overloaded
  - Variables can be sent to functions by value, using a pointer and by reference
  - A call by reference can be used without sending a pointer, but rather sending the objects address without the pointer mediator.

Reference variable

- Reference type can be thought as alternative name for a variable (object).
- Reference must be initialized at definition time.
- **Example**:

```
long index = 10 ;
long &refIndex = index ;
cout<< ++refIndex << endl;
cout<< index << endl;
long &refIndex2 ;        //ERROR must be initializate
long &refIndex3 = 10 ;   //ERROR int to long &
```

- Output:
  - 11
  - 11

Call function by reference


```
#include <iostream>
#include <string>
void square(int &num){
  num = (num) * (num);
}
void main() {
int x = 5;
std::cout << x << "   ";
square(x);
std::cout << x << std::endl;
}
```

Using pointer


```
#include <iostream>
#include <string>
void square(int  * num){
  *num = (*num) * (*num);
}
void main() {
int x = 5;
std::cout << x << "   ";
square(&x);
std::cout << x << std::endl;
}
```

C++ vs. Java

The definition of classes in C++ is somewhat different than in Java.

Here is an example: a C++ version of the Point class:

```
class Point{
public:
   Point();
   Point(double xval, double yval);
   void  move(double dx, double dy);
   double  getX() const;
   double  getY() const;
private:
   double  x;
   double  y;
};
```

Several essential differences.

1. In C++, there are public and private *sections*, started by the keywords public and private. In Java, each individual item must be tagged with public or private.

2. The class definition can contains only the declarations of the methods. The actual implementations can be listed separately.

3. There is a semicolon at the end of the class.

Diff (cont.)

4. The major difference between Java and C++ is the behavior of object variables.

   In C++, object variables hold values, not object references.

   Note that the new operator is never used when constructing objects in C++. You simply supply the construction parameters after the variable name.

   Example :

```
Point  p(1, 2); /* construct p */
```

Diff (cont.)

If you do not supply construction parameters, then the object is constructed with the default constructor.

```
Time now; /* construct now with Time::Time() */
```

This is very different from Java.

In Java, this command would merely create an uninitialized reference.

In C++, it constructs an actual object.

Diff (cont.)

When one object is assigned to another, a copy of the actual values is made.

In Java, copying an object variable merely establishes a second reference to the object. Copying a C++ object is just like calling clone in Java. Modifying the copy does not change the original.

```
Point q = p; /* copies p into q */
q.move(1, 1); /* moves q but not p */
```

Example

The implementation of methods follows the class definition.

Because the methods can be defined outside the classes, each method name is prefixed by the class name.

The :: operator separates class and method name.

Accessor methods that do not modify the implicit parameter are tagged as const.

```
Point::Point() { x = 0; y = 0; }
void Point::move(double dx, double dy) {
  x = x + dx;
    y = y + dy;
}
double Point::getX() const{
  return x;
}
```

Building C++ program

- Each class is defined in a separate header (.h) file
- Each class implementation is carried out in a separate .cpp file.
- The program implementation is carried out in a different separate .cpp file.

Preprocessor: #ifdef and #ifndef

- The #ifdef (if defined) and #ifndef (if not defined) preprocessor commands are used to test if a preprocessor variable has been "defined".
- The common use for this is **Prevent multiple definitions in header files**
- #ifndef checks whether the given token has been #defined earlier in the file or in an included file; if not, it includes the code between it and the closing #else or, if no #else is present, #endif statement. #ifndef is often used to make header files .
- When there definitions in a header file that can not be made twice, the code below should be used. A header file may be included twice other include files include it, or an included file includes it and the source file includes it again.
- To prevent bad effects from a double include, it is common to surround the body in the include file with the following (where MYHEADER_H is replaced by a name that is appropriate for your program).

```
#ifndef MYHEADER_H
#define MYHEADER_H
. . .
                    // This will be seen by the compiler only once
#endif /* MYHEADER_H */
```

Preprocessor: #ifdef and #ifndef

- An identifier must follow the **#ifndef** keyword. The following example defines MAX_LEN to be 50 if EXTENDED is not defined for the preprocessor. Otherwise, MAX_LEN is defined to be 75.

```
#ifndef EXTENDED
 #define MAX_LEN 50
#else
 #define MAX_LEN 75
#endif
```

These directives check only for the presence or absence of identifiers defined with **#define**, not for identifiers declared in the C or C++ source code.

Example string.h - a String class


```
#ifndef STRING_H
#define STRING_H
#include <cstring>
#include <iostream>
using namespace std;
class String {
         enum {SIZE=256};
         char   m_str[SIZE];
         int    m_len;
public:
         String() {strcpy(m_str,""); m_len=0; }
         String(char *s) {strcpy(m_str, s); m_len=strlen(m_str);}
         int    len() { return m_len;}
         void concat(String &s){strcat(m_str, s.m_str); m_len += s.m_len;}
         bool equals(String &s) {return strcmp(m_str, s.m_str)==0;}
         void print(){cout<< "m_str="<< m_str<< ", m_len="<< m_len<<endl;}
         void read_word();
         void read_line(); };
#endif
```

string .cpp


```
#include <iostream>
#include <cstring>
#include "string.h"
using namespace std;
// read 1 word into the string and test for errors in input
void String::read_word() {
        cout << "Enter a word:";
        cin >> m_str;
        if(cin)
                m_len = strlen(m_str);
        else            {
                strcpy(m_str, "");
                m_len = 0;
            }
}
```

string .cpp


```
// read a line of text into the string and test for errors
  in input

void String::read_line() {
  cin.ignore(256, '\n'); // clear current line
  cout << "Enter a line of text:";
  cin.getline(m_str,SIZE, '\n');
  if(cin)
             m_len = strlen(m_str);
  else    {
             strcpy(m_str, "");
             m_len = 0;
          }
}
```

string_main.cpp - the using program


```
#include "string.h"
using namespace std;
int main() {
  String s1("hello"), s2("world"),
  s3;
  s3 = s1;
```

```
   s1.concat(s2);
  String s4;
   s4.read_word();
  String s5;
  s5.read_line();
  s1.print();
  s2.print();
  s3.print();
  s4.print();
  s5.print();
}
```