Introduction to C++

Himanshu Singh

Github

Recap of C

Built-in/Basic Data types in C

- char
- int
- float
- double

Additionally, C89 define _Bool

void : it indicates no type

Derived data types

- Array
- Structure – struct and union
- Pointer
- Function
- String – C String are not a data type, but can me made to behave as such using <string.h>

Type Modifiers:

- short
- long
- signed
- unsigned

Literals refer to fixed values of built-in type. Literal can be of any of the basic data types

212	//(int) Decimal literal
0173	//(int) Octal literal
0b11001	//(int) Binary literal
0xF23	//(int) Hexadecimal literal
3.14	//(double) Floating-point literal
‘x’	//(char) Character literal
“Hello”	//(char*) String literal

An operator denotes specific operation.

- Arithmetic Operators + - * / % ++ --
- Relational Operators== != > < >= <=
- Logical Operators && || !
- Bitwise Operators | & ~ ^ << >>
- A1ssignment Operators = += -= *= ...
- Miscellaneous Operators . , sizeof & ?:

Precedence of all operator: (), [], ++, --, +(unary), -(unary), !, ~, *(pointer ref), &(address of), sizeof, *, /, %. +, -, <<, >>, ==, !=, *=, =, /=, &, |, &&, ||, ?:, =, +=, -=, *=, /=, <<=, >>=

Structure is collection of data items of different types. Data items are called members.

```
typedef struct _complex {
    double re;
    double im;
} Complex;
Complex c = {2.0, 3.5};
```

Union is special structure that allocates memory only for the largest data member and hold only one member at a time.

```
typdef union _packet {
    int     iData;
    double  dData;
    char    cData;
} Packet;
Packet p = {10};
```

Pointer Array Duality

```
int a[] = {1, 2, 3, 4, 5};
int *p;
p = a;
printf("a[0] = %d\n", *p);
printf("a[1] = %d\n", *++p);
printf("a[3] = %d\n", *(p+2));
```

Pointer to structure

```
typedef struct _complex {
    double re;
    double im;
} Complex;
Complex c = {2.3, 4.7};
complex *p = &c;
(*p).re = 6.3;
p->im = 3.1;
```

C++ Intro

Hello World

```
#include <iostream>
int main() {
    std::cout << “Hello World in C++”;
    std::cout << std::endl;
    return 0;
}
```

IO header is iostream.
Opeartor << to stream to console, is a binary operator

Add two numbers

```
#include<iostream>
int main() {
    int a, b;
    std::cout << “Input two numbers\n”;
    std::cin >> a >> b;
    int sum = a + b;
    std::cout << “Sum of ” << a << “ and ” << b << “ is: ” << sum << std::endl;
}
```

operator >> to stream from console, is a binary operator.

Square root of number

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    cout << “Input number : ” << endl;
    cin >> x;
    double sqrt_x = sqtr(x);
    cout << “Square root of ” << x << “ is: ” << sqrt_x << endl;
}
```

Math header is cmath (C standard Library in C++), sqrt function is from this library

In C++ Standard Library, all names are in std namespace (std::cout, std::cin). We use using namespace std to get rid of std:: for every standard library name.

C/C++ Header Conventions

	C Header	C++ Header
C Program	Use .h Example: #include <stdio.h> Names in global namespace	Not Applicable
C++ Program	Prefix c and no .h Example: #include <cstdio>	No .h Example: #include <iostream>

Dynamically managed array size

```
#include<iostream>
#include<vector>
using namespace std;
int main() {
    vector<int> arr;
    cout << “Enter the  number of elements ”;
    int count, sum = 0;
    cin >> count;
    arr.resize(count);
```

```
    for(int i = 0; i < count; i++) {
        arr[i] = i;
        sum += arr[i];
    }
    cout << "Array sum : " << sum << endl;
}
```

resize fixed vector size at run time. Whereas in C for dynamically managing the size of array, we need functions like malloc, calloc and free.

Stings in C and C++

In C, strings are in string.h library. They are array of char terminated by NULL. C-String is supported by functions in string.h in C standard library.

In C++, string is a type. With opeartors (like + for concatenation) it behaves like a built-in type. In addition, for functions in C standard library string.h can be used in C++ as cstring in std namespace.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str1 = "Hello ";
    string str2 = "World";
    string str = str1 + str2;
    cout << str;
}
```

Function	Description (Member Function)
Member Function	
Constructor	Construct string object (public)
Destructor	String destructor (public)
Operator =	String Assignment (public)
Iterators	
begin	Return iterator to beginning (public)
end	Return iterator to end (public)
rbegin	Return reverse iterator to reverse beginning (public)
rend	Return reverse iterator to reverse end (public)
cbegin	Return const_iterator to beginning (public)
cend	Return const_iterator to end (public)
crbegin	Return const_reverse_iterator to reverse beginning (public)
crend	Return const_reverse_iterator to reverse beginning (public)

More about std::string

Sorting and Searching

```
#include<iostream>
#include<algorithm>
using namespace std;
bool compare(int i, int j)
{
    return (i>j);
}
int main()
```

```
{
    int data[] = {34,76,12,45,2};
    sort(data, data+5, compare);
}
```

Binary Search

```
#include<iostream>
#include<algorithm>
using namespace std;
int main()
{
    int data[] = {2,5,9,34,65,78};
    int k = 65;
    if(binary_search(data, data+5, k))
        cout <<  “Found!” << endl;
    else
        cout << “Not Found!” << endl;
}
```

const

The value of const variable cannot be changed after definition, and it must be initialized when defined.

```
const int n = 10;
n = 5;              //Compilation Error, n cannot be changed.
int *p  = 0;
p =  &n;            //Compilation error, as n may be changed by *p
```

Manifest constant	Constant value
Is not type safe	Has its type
Replaced textually in cpp	Visible to compiler
Cannot be watched in debugger	Cane be watched in debugger
Evaluated as many times as replaced	Evaluated only on initialization

const-ness can be used with Pointers in 2 ways:

- pointer to constant data, where pointee (pointed data) cannot be changed.
- Constant Pointer, where pointer (address) cannot be changed

---

```
int m = 4;
const int n = 5;
const int * p = &n;
n = 6;                    //Error, n is constant and cannot be changed
*p = 7;                   //Error, p points to const data, cannot be changed.
p = &m;                   //Okay

const int i = 5;
int * j = &i;                   //Error
```

```
int m = 4, n=4;
int * const p = &n;
n = 6;                    //Okay
*p = 7;                   //Okay
p = &m;                   //Error, p is const pointer and cannot be chnaged.
```

```
const in m = 4;
const int n = 4;
const int * const p = &n;
n = 6;                          //Error n is constatnt
*p = 7;                         // Error, p points to const data
p = &m;                         //Error, p is const pointer
```

```
To decide constness, draw a mental line through *
int n = 5;
int * p = &n;              //non-constant pointer to non-constant pointee
const int * p = &n;        //non-constant pointer to constant pointee
int * const p = &n;        //constant pointer to non-constant pointee
const int * const p = &n;  //constant pointer to constant pointee
```

inline function: just a function like any other. The function prototype is preceded with keyword inline. An inline function is expanded at the site of its call and the overhead of passing parameter of caller and callee function is avoided.

Reference: An alias/synonym for an existing variable. They have same memory location, changing one changes the another.

```
int i = 10;
int &j = i;  //j is reference to i
```

Wrong Declaration	Reason	Correct Declaration
int& i;	No variable (address) to refer to, must be initialized.	int& i = j;
int& j = 5;	No address to refer to as 5 is constant	const int& j = 5;
int& i = j + k;	Only temporary address	const int& i = j +k;

Call by reference

```
void swap(int& x , int& y)
{
    int t = x;
    x = y;
    y = t;
}
void main()
{
    int a =10, b =5;
    swap(a,b);
}
```

A reference parameter may be changed in the function. We can use const to stop reference parameter being changed.

```
int ref_const (const int& x)
{
    return x+1;
}
int main()
{
    int a =10, b;
    b = ref_const(10);
    return 0;
}
```

C++ allows programmer to assign default values to function parameters. Default parameters are specified while prototyping the function.

All parameters to the right of parameter with default argument must have default argument.

Default arguments cannot be redefined. All non-default parameters are needed in a call.

```
void f(int, double = 0.0, char*);          //Error missing default parameter for
                                             parameter 3
void g(int, double = 0.0, char* =  NULL);  //Okay
```

Function Overloading/ Static Polymorphism: functions with same name, similar functionality but different algorithm and identified by different interface data types.

```
typedef struct {int data[10][10];}  Mat;
typedef struct {int data[1][10];}   vecRow;
typedef struct {int data[10][1];}   vecCol;

void Multiply(const Mat& a,    const Mat& b,    Mat& c);
void Multiply(const vecRow& a, const Mat& b,    vecRow& c);
void Multiply(const Mat& a,    const vecCol& b, vecCol& c);
void Multiply(const vecCol& a, const vecRow& b, Mat& c);
void Multiply(const vecRow& a, const vecCol& b, int& c);
```

Two functions with same signature but different return types cannot be overloaded.

```
int    Area (int a, int b) { return a*b; }
double Area (int a, int b) { return a*b; }   //ERROR
```

Overload resolution:

- Identify the set of candidate functions.
- From the set of candidate function, identify set of viable functions.
- Select best viable function
  - Exact
  - Promotion
  - Standard type conversion
  - User defined type conversion

Exact: lvalue to rvalue conversion, Array to Pointer conversion, Function to pointer conversion
Promotion: char to int; float to double; enum to int/short/unsigned int; bool to int
Standard conversion: integral type conversion, less precise floating to more precise, conversion from integral to floating point types, boo conversion.

Operator Functions Implicit for predefined operators of builtin types and cannot be redefined.

Operator Expression	Operator Function
a + b	operator+(a, b)
a = b	operator=(a, b)
c = a + b	operator=(c, operator+(a, b))

Operator overloading (ad-hoc polymorphism): different operators have different implementations depending on their argument.
Intrinsic properties of overloaded operator cannot be changed. (arity, precedence, associativity)
For unary prefix, use MyType& operator++(MyType& s1)
For unary postfix, use MyType& operator++(MyType& s1, int)
operator:: (scope), operator. (member access), operator.* (member access through pointer) operator sizeof and operator?: (ternary conditional) cannot be overloaded.
The overloads of operator&&, operator||, operator, (comma) lose their special proprieties.
The overload-> must return a raw pointer or object reference.

OOP in C++

A class is an implementation of a type, contains data members/ attributes and offer data abstraction/ encapsulation of Object Oriented Programming. Classes also provide access specifier for members to enforce data hiding that separate implementation from interface.

An object of class is an interface created according to its blue print. It comprises data members that specify its state, also support member functions that specify its behaviour.

An implicit this pointer holds the address of an object. Type of this pointer for class X:

X * const this;

this pointer is accessible only in member functions.

```
#include<iostream>
using namespace std;
class X{
public:
    int m1,m2;
    void f(int k1, int k2)
    {
        m1 = k1;                         //Implicit access without this pointer
        this->m2 = k2;                   //Explicit access with this pointer
        cout << "Id = " << this << endl;
    }
};
int main()
{
    X a;
    a.f(2,3);
    cout << "Addr = " << &a << endl;
    return 0;
}
```

this pointer is implicit passed to methods.

Classes provide access specifiers for members (data as well as function) to enforce data hiding that separates implementation from interface.

private: accessible inside definition of class.

public: accessible everywhere.

Information Hiding: The private part of class form its implementation because the class alone should be concerned with it and have the right to change it. The public part of a class constitute its interface which is available for all other for using the class.

Customarily, we put attribute in private part and member functions in public part. This ensures:

- The state of object can be changed only through one of its member functions
- The behaviour of an object is accessible to other through member functions.

Classes wrap data and function acting on the data together as a single data structure. This is Aggregation. Access specifier defines the visibility outside the class, this helps in hiding information about the implementation details of the data members and method. This concept is known as Encapsulation.

Constructor & Destructor

Constructor is implicitly called at instantiation as set by the compiler. Destructor is implicitly called at end of scope. If user has not provided any constructor or destructor, compiler provides free default constructor and destructor, they have no code in the body.

```
#include<iostream>
using namespace std;
class Stack{
private:
    char *data_; int top_;
public:
    Stack() : data (new char[10]), top_(-1) {}
    int empty() { return (top == -1); }
    void push(char x) { data[++top_] = x; }
    void pop() { top_--; }
    char top() { return data[top_]; }
    ~Stack() { delete [] data; }
};
int main()
{
    Stack s;
    for(int i = 0; i < 5; i++)
        s.push('A'+ i);
    while(!s.empty())
    {
        cout << s.top();
        s.pop();
    }
}
```

Constructor	Member Functions
• Is a static member function without this pointer.	• Has implicit this pointer
• Name is same as name of class.	• Any name different from name of class.
• Has no return type, not even void.	• Must have at least one return type.
• Implicit call by instantiation	• Explicit call by the object.

Destructor	Member Functions
• Has implicit this pointer.	• Has implicit this pointer
• Name is ~ followed by name of class.	• Any name different from name of class.
• Has no return type, not even void and does not return anything.	• Must have at least one return type and a return statement.
• Implicit call by instantiation	• Explicit call by the object.

Order of initialization in constructor does not depend on the order in the initialization list. It depends on the order of data members in the definition.

Fail Scenario

```
#include<iostream>
#include<cstring>
#include<cstdlib>
using namespace std;
class String{
    size_t len_; char *str_;
public:
    String(char *s) : str_(strdump(s)), len_(strlen(str_))
    {
        cout << "Ctor";
```

```
        print();
    }
    ~String()
    {
        cout << "Dtor";
        print();
        free(str_);
    }
    void print(){
        cout << "(" << str_ << ": " << len_ << ")" << endl;
    }

};
int main()
{
    String s = "C++";
    s.print();
    return 0;
}
```

Here len_ precedes str_ in list of data member, len(strlen(str_)) is executed before str_(strdump(s)). When strlen(str_) is called str_ is still uninitialized, which is a garbage value, May also cause program to crash.

Copy Constructor

```
#include<iostream>
#include<cmath>
using namespace std;
class Complex{ double re_, double im_;
public:
    Complex(double re, double im) : re_(re)), im_(im)
    {
        cout << "Ctor : "; print();
    }
    Complex(const Complex& c) : re_(c.re_)), im_(c.im_)
    {
        cout << "Copy Ctor : "; print();
    }
    ~Complex()
    {
        cout << "Dtor : "; print();
    }
    double norm() { return sqrt(re_ * re_ + im_ * im_); }
    void print(){
        cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl;
    }

};
int main()
{
    Complex c1(4.2,5.4), c2(c1);
}
```

Ctor : |4.2+j5.4| = 6.8410
Copy Ctor : |4.2+j5.4| = 6.8410
Dtor : |4.2+j5.4| = 6.8410

Dtor : |4.2+j5.4| = 6.8410

If no copy constructor is provided by the user, the compiler supplies a free one. Free copy constructor cannot initialize the object to proper values. It perform Shallow Copy.
Shallow Copy (bitwise copy): An object is created by simply copying the data of all variables of original object, works well if none of the variables are defined in heap/ free store. For dynamically crated variables, the copied objects refers to the same memory location. This creates ambiguity.
Deep Copy (Lazy Copy) An object is created by copying data of all variables except the ones on heap. Allocates similar memory resources with same value to the object. Need to explicitly define copy constructor and assign dynamic memory as required.

Copy Assignment operator

```
#include<iostream>
#include<cmath>
using namespace std;
class Complex{ double re_, double im_;
public:
    Complex(double re, double im) : re_(re)), im_(im)
    {
        cout << “Ctor : ”; print();
    }
    Complex(const Complex& c) : re_(c.re_)), im_(c.im_)
    {
        cout << “Copy Ctor : ”; print();
    }
    Complex& opearator=(const Complex& c) : re_(c.re_)), im_(c.im_)
    {
        cout << “Copy Assignemnt : ”; print();
        return *this;           //return *this for chaining
    }
    ~Complex()
    {
        cout << “Dtor : ”; print();
    }
    double norm() { return sqrt(re_ * re_ + im_ * im_); }
    void print(){
        cout << “|” << re_ << “+j” << im_ << “| = ” << norm() << endl;
    }

};
int main()
{
    Complex c1(4.2,5.4), c2(5.8,3.8), c3(c2);
    c3.print();
    c3 = c1;
    c3.print();
}
```

Ctor : |4.2+j5.4| = 6.8410
Ctor : |5.8+j3.8| = 6.9339
Copy Ctor : |5.8+j3.8| = 6.9339
|5.8+j3.8| = 6.9339
|4.2+j5.4| = 6.8410

Dtor : |4.2+j5.4| = 6.8410
Dtor : |5.8+j3.8| = 6.9339
Dtor : |4.2+j5.4| = 6.8410

```
#include<iostream>
#include<cstring>
#include<cstdlib>
using namespace std;
class String{ char *str_, size_t len;
public:
    String(char *s) : str_(strdump(s)), len_(strlen(str_)) { }
    String(const String& s) : str_(strdump(s.str_)), len_(s.len_) { }
    String& opeartor=(const String& s)
    {
        if(this != &s){                //check for self copy
            free(str);
            str_ = strdup(s.str_);
            len_ = s.len_;
            return *this;
        }
    }
    ~String() { free(str_); }
    void print(){
        cout << "(" << str_ << ": " << len_ << ")" << endl;
    }

};
int main()
{
    String s1 = "Football", s2 = "Cricket";
    s2.print();
    s2 = s1;
    s2.print();
    return 0;
}
```

(Cricket : 7)
(Football: 8)

"this != &s" comes to check if there is self copy, if there is no check, it will free the variable and behaviour of that variable will be ambiguous.

const-ness

constant objects User defined Data types can also be made constant, if an object is constant none of the data members can be changed. The type of this pointer of constant object of class is:
const MyClass * const this;
A constant objects cannot invoke normal methods of class lest these methods change the object.

```
#include <iostream>
using namespace std;
class MyClass{
    int privmember;
public:
```

```
    int pubmember;
    MyClass(int pri, int pub) : privmember (pri) , pubmember(pub) {}
    int getmember() { return privmember; }
    void setmember(int i) {privmember = i; }
    void print() { cout << privmember << " , " << pubmember << endl; }
};
int main()
{
    const MyClass myconstobj(5,6);
    cout << myconstobj.getmember() << endl;         //Error1
    myconstobj.setmember(7);                        //Error2
    myconstobj.pubmember  = 8;                      //Error3
    myconstobj.print();                             //Error4
}
```

It is not allowed to invoke methods or make changes in constant object myconstobj.

constant Member Function To declare a constant member function, we use the keyword const between function header and body. It expect this pointer as const MyClass * const this;

Also, no data member can be changed.

Non-constant objects can invoke constant member functions and non-constant member function.

Constant objects can only invoke constant member function.

```
#include <iostream>
using namespace std;
class MyClass{
    int privmember;
public:
    int pubmember;
    MyClass(int pri, int pub) : privmember (pri) , pubmember(pub) {}
    int getmember() const { return privmember; }
    void setmember(int i) {privmember = i; }
    void print() const { cout << privmember << " , " << pubmember << endl; }
};
int main()
{
    MyClass myobj(1,2);
    //non-const object can invoke all member functions
    cout << myobj.getmember() << endl;
    myobj.setmember(7);
    myobj.pubmember  = 8;
    myobj.print();
    const MyClass myconstobj(5,6);
    //const object cannot allow any change
    cout << myconstobj.getmember() << endl;
    //myconstobj.setmember(7);                  //can't invoke non-const member func
    //myconstobj.pubmember  = 8;                //cant update data member
    myconstobj.print();
}
```

constant Data Members A constant data member cannot be changed even in non-constant object, and must be initialized on the initialization list.

```
#include <iostream>
using namespace std;
class MyClass{
    const int cprimem;
```

```
    int ncprimem;
public:
    const int cpubmem;
    int ncpubmem;
    MyClass(int cpri, int ncpri, int cpub, int ncpub): cprimem(cpri),
                          ncprimem(ncpri), cpubmem(cpub), ncpubmem(ncpub) {}
    int getcpri() { return cprimem; }
    void setcpri(int i) { cprimem = i; }     //Error: Assignment to const data
    int getncpri() { return ncprimem; }
    void setncpri(int i) { ncprimem = i; }
};
int main()
{
    MyClass myobj(1,2,3,4);
    cout << myobj.getcpri() << endl;
    cout << myobj.getncpri() << endl;
    myobj.setncpri(7);
    cout << myobj.cpubmem << endl;
    //myobj.cpubmem = 10;              //Error: Assignment to const data
    cout << myobj.ncpubmem() << endl;
    myobj.cpubmem= 20;
}
```

mutable Data Members mutable data member is changeable in a constant object. mutable is provided to model Logical (Semantic) const-ness against the default bit-wise (syntactic) const-ness of C++

- mutable is applicable only yo data members and only to variables
- Reference data members cannot be declared mutable
- static data members cannot be declared mutable
- const data members cannot be declared mutable

```
#include <iostream>
using namespace std;
class MyClass{
    int mem_;
    mutable int mutmem_;
public:
    MyClass(int m, int mm): mem_(m), mutmem_(mm){}
    int getmem() const { return mem_; }
    void setmem(int i) { mem_ = i; }
    int getmutmem() const { return mutmem_; }
    void setmutmem(int i) const { mutmem_ = i; }
};
int main()
{
    const MyClass myconstobj(1,2);
    cout << myconstobj.getmem() << endl;
    //myconstobj.setmem(3);
    cout << myconstobj.getmutmem () << endl;
    myconstobj.setmutmem(10);
}
```

static

static data members is associated with class not object, is shared by all objects of a class.

- Need to be defined outside the class scope to avoid linker error.
- Must be initialized in source file
- is constructed before main() starts and destructed after main() ends.
- Can be accessed
  - with class-name followed by the scope resolution (::)
  - as a member of any object of the class

Order of initialization of static data members does not depend on their order in the definition of the class. It depends on the order their definition and initialization in the source.

static member functions

- does not have this pointer - not associated with ant object
- cannot access non-static data members
- cannot invoke non-static member functions
- may initialize static data members even before any object creation
- cannot be declared as const

```
#include <iostream>
using namespace std;
class PrintJobs{
    int nPages_;
    static int nTrayPages_;
    static int nJobs_;
public:
    PrintJobs(int nP): nPages_(nP){
        ++nJobs;
        cout << “Printing ” << nP << “ Pages” << endl;
         nTrayPages_ -= nP;
        }
    ~PrintJobs(){--nJobs;}
    static int getjobs() { return nJobs_; }
    static int checkpages() { return nTrayPages_; }
    static void loadpages(int nP) { nTrayPages_ += nP; }
};
int PrintJobs::nTrayPages_ = 500;
int PrintJobs::nJobs_ = 0;
int main()
{
    cout << “Pages = ” << PrintJobs::checkpages() << endl;
    PrintJobs job1(10);
    cout << “Jobs = ” << PrintJobs::getjobs() << endl;
    cout << “Pages = ” << PrintJobs::checkpages() << endl;
    {
        PrintJobs job1(30), job2(20);
        cout << “Jobs = ” << PrintJobs::getjobs() << endl;
        cout << “Pages = ” << PrintJobs::checkpages() << endl;
        PrintJobs::loadpages(100);
    }
    cout << “Jobs = ” << PrintJobs::getjobs() << endl;
    cout << “Pages = ” << PrintJobs::checkpages() << endl;
}
```

Jobs = 0

Pages = 500

Printing 10 pages

Jobs = 1
Pages = 490
Printing 30 pages
Printing 20 pages
Jobs = 3
Pages = 440
Jobs = 1
Pages = 540

Static Member function	Non-static Member function
• Cannot access non-static data members or methods • Can be invoked anytime during program execution • cannot be virtual or constant • Constructor is static though not declared static	• Can access non-static data members and methods • Can be invoked only during lifetime of the object • May be virtual or constant • There cannot be non-static constructor.

Singleton Class is a creational design pattern satisfying :

- ensure that only one object of its kind exists
- provides single point of access.

Meyer's Singleton

```
class Printer{
    bool BlackAndWhite, BothSide;
    Printer(bool bw = false, bool bs = false) : BlackAndWhite(bw),BothSide(bs){
        cout << "Printer Constructed!!" << endl;
    }
    ~Printer(){
        cout << "Printer Destructed!!" << endl;
    }
public:
    static const Printer& printer(bool bw = false, bool bs = false){
        static Printer myPrinter(bw,bs);
        return myPrinter;
        }
    void print(int nP) const{
        cout << "Printing " << nP << " Pages" << endl;
    }
};
int main()
{
    Printer::printer().print(10);
    Printer::printer().print(12);

}
```

friend function

A friend function of a class has access to private and protected members of the class (breaks the encapsulation) in addition to public members.

- does not have name qualified with class scope

- is not called with an invoking object of the class
- can be declared friend in more than one class.

A friend function can be a

- global function
- a member function of a class
- a function template

```
#include<iostream>
using namespace std;
class MyClass{
    int data_;
public:
    MyClass(int i) : data_(i) { }
    friend void dispaly (const MyClass& a);
};
void dispaly (const MyClass& a) {
    cout << “data = ” << a.data_;
}
int main()
{
    MyClass obj(10);
    dispaly(obj);
}
```

friend class has access to the private and protected members of the class in addition to piblic members. It can be declared friend in more than one class.

Inheritence

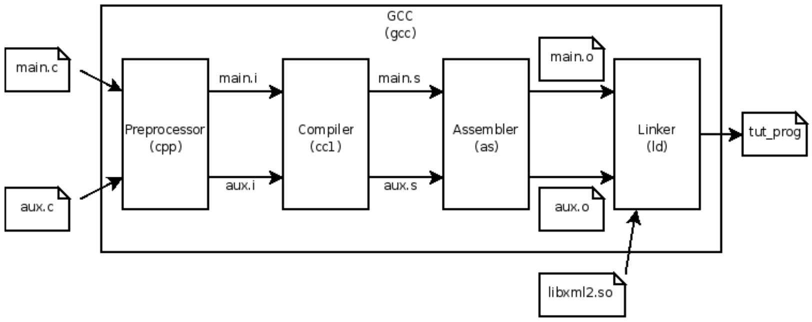
Polymorphism

Exceptions and Telmplates

STL

C++11 and beyond

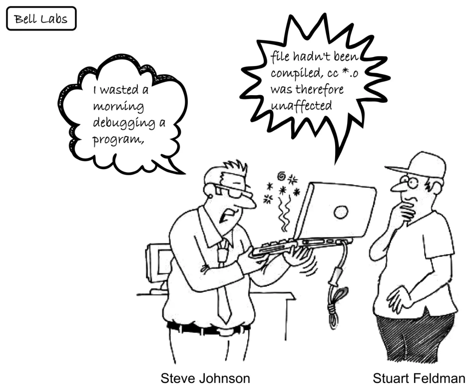
Build Pipeline



The C preprocessor (CPP) has the ability for the inclusion of header files, macro expansion, conditional compilation and line control/ It works on .c .cpp .h files and produces .i files.
The compiler translates the preprocessed C/C++ code into assembly language which is a machine level code in text that contains instructions that manipulates the memory and processor directly. It works on .i files and produces .s files.
The Assembler translates the assembly file into binary machine language or object code it works on .s files and produces .o files.
The linker links are program with the pre-compiled libraries for using their functions and generates the executable binary. It works on .o (static library) .so (shared library or dynamically linked library) and .a (library archive) files and produce a.out file.

make

make is tool which controls the generation of executables and other non-source files of a program from the programs's source files.
The manual process iof build would be difficult in any practical software project due to:



- Volume: project has hundreds of folder, source and header files, need hundreds of command
- Workload: Build needs to be repeated several times a day with code changes in some file
- Dependency: Often change in one file, all translation units do not need to be recompiled
- Diversity: We may need to use different tools for different files, different build flags, different folder structures.

```
hello: hello.c main.c
	gcc -o hello hello.c main.c -I
```

Write these lines in a text file named makefile or Makefile and run make command and it will execute the command. Make file comprises a number of rules. Every rule has a target to build, a colon separator(:). Zero or more files on which the target depends on and commands to build on the next line. Hash(#) starts a comment that continue till the end of the line.

<u>Simple and recursive variable</u>

Simply expanded variables (defined by :=) are evaluated as soon as encountered
Recursively expanded variables (defined by =) are lazily evaluated, may be defined after use.

Simply Expanded	Recursively Expanded

MAKE_DEPEND := $(CC) -M
...
#some time later
CC = gcc
	MAKE_DEPEND = $(CC) -M
	...
	#some time later
	CC = gcc

$(MAKE_DEPEND) expands to:	
<space>-M	ccc -M