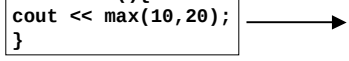
Inline Vs. Out of line code

- In C++, you can ask the compiler to create short functions that are not actually called.
- Instead, their code is expanded in line at the point of each invocation.
- You cause ask the compiler to expand a function inline by preceding its definition with the **`inline`** keyword:

```
inline int max(int a, int b){
   if (a>b)
     return 1;
   else
     return 0;
}
```


Inline Vs. Out of line code

```
void main(){
cout << max(10,20);
}
```



```
void main(){
if (a>b)
     result=1;
   else
     result=0;
cout << result;
}
```

- Inline functions can be efficient because:
  - when functions are called, a significant amount of overhead is generated by the calling and return mechanism.
  - when a function is expanded inline, none of these operations occur, producing faster runtimes, mostly.

Inline Vs. Out of line code

- But:
  - Code size is larger because of the duplicated code, resulting in cache misses and longer load times
  - Therefore it is best to inline only very small functions, or functions with few call sites
  - It is also better to inline only those functions which will have significant impact on the performance of your program
  - Whether there is actually any speedup (or slowdown) depends on many factors, including the compiler, possibilities for other optimisations, the shape of the call graph, the instruction cache and many other things.
  - The inline keyword in C++ is only a request to the compiler. The compiler is free to (and often does) ignore the request if it thinks it a bad idea.

Inline functions in a class

- You may define short functions within a class declaration
- When a function is defined inside a class definition:
- ***it is automatically made into an inline function.***
- It is not necessary to precede its declaration with the **`inline`** keyword.


```
class myclass {
   int a,b;

public:
   //automatic inline
   void init(int i, int j){a=i; b=j;}
   void show() {cout<< a <<" " << b<< "\n";}
};

void main()
{
   myclass x;
   x.init(10,20);
   x.show();
}
```


Overloading of functions

- Overloading is a process which allows several items providing different facilities, to have the same name.
- The compiler chooses the appropriate item to use.

Different number of parameters

Consider the following two versions of function **Larger()**:

```
#include <iostream.h>
int Larger(int, int);
int Larger(int, int, int);
```

```
int Larger(int a,int b){
if (a > b)
  return a;
else
  return b;
}
```

```
int Larger (int a,int b,int c){
if (Larger(a,b) > c)
   return Larger(a,b);
else
   return c ;
}
```


Different number of parameters

Depending on the number of parameters, the correct version of `Larger()` will be called:

```
void main(){
cout<< "The larger of 2 and 3 is"<<Larger(2,3);
cout<< "\n and of 4, 5 and 6 is"<<Larger(4,5,6);
}
```


Different Types of Parameters

```
#include <iostream.h>
class Num{
private:
   int iNum;
   float fNum;
   double dNum;
public:
   void Init(int i){iNum = i;
                    cout << "integer number\n";}
   void Init(float f){fNum = f;
                    cout << "float number\n";}
   void Init(double d){dNum = d;
                    cout << "double number\n";}
};
```


Different Types of Parameters

```
void main()
{
int ix;
float fy;
double dz;

Num myNum;

myNum.Init(ix);
myNum.Init(fy);
myNum.Init(dz);

}
```

OUTPUT:

```
integer number
float number
double number
```


Default values to parameters

- If a default value is given to a parameter, then it may be omitted on an invocation.
- The only restriction is that if a parameter has a default value, then all parameters to the right of it must also have a default value.

Default values to parameters

```
int sum (int , int = 0, int = 0 ); // Prototype

int sum (int a, int b, int c){
      return a + b + c;             // Implementation
}


void main(){                        // Usage

 cout << "Sum of 10 and 20 is: " << sum(10,20);
 cout << "\nSum of 10, 20 and 30 is: "<< sum(10,20,30);
}
```