在 C++语言的设计中，内联函数的引入可以说完全是为了性能的考虑。因此在编写对性能要求比较高的 C++程序时，非常有必要仔细考量内联函数的使用。

所谓“内 联”，即将被调用函数的函数体代码直接地整个插入到该函数被调用处，而不是通过 call 语句进行。当然，编译器在真正进行“内联”时，因为考虑到被内联函 数的传入参数、自己的局部变量，以及返回值的因素，不仅仅只是进行简单的代码拷贝，还需要做很多细致的工作，但大致思路如此。

开发人员可以有两种方式告诉编译器需要内联哪些类成员函数，一种是在类的定义体外；一种是在类的定义体内。

（1）当在类的定义体外时，需要在该成员函数的定义前面加“inline”关键字，显式地告诉编译器该函数在调用时需要“内联”处理，如：

```
class Student

{

public:

    String GetName();

    int    GetAge();

    void       SetAge(int ag);

    ……

private:

    String  name;

    int    age;

    ……

};

inline String GetName()
```

```
{
    return name;
}
inline int GetAge()
{
    return age;
}
inline void SetAge(int ag)
{
    age = ag;
}
```

（2）当在类的定义体内且声明该成员函数时，同时提供该成员函数的实现体。此时，“inline”关键字并不是必需的，如：

```
class Student
{
public:
    String GetName()    { return name; }
    int    GetAge()     { return age; }
    void   SetAge(int ag) { age = ag; }
    ……
private:
```

```
    String  name;
    int     age;
    ……
};
```

当普通函数（非类成员函数）需要被内联时，则只需要在函数的定义时前面加上“inline”关键字，如：

```
inline int DoSomeMagic(int a, int b)
{
    return a * 13 + b % 4 + 3;
}
```

因为 C++是以 “编译单元”为单位编译的，而一个编译单元往往大致等于一个“.cpp”文件。在实际编译前，预处理器会将“#include”的各头文件的内容（可能会 有递归头文件展开）完整地拷贝到 cpp 文件对应位置处（另外还会进行宏展开等操作）。预处理器处理后，编译真正开始。一旦 C++编译器开始编译，它不会意 识到其他 cpp 文件的存在。因此并不会参考其他 cpp 文件的内容信息。联想到内联的工作是由编译器完成的，且内联的意思是将被调用内联函数的函数体代码直 接代替对该内联函数的调用。这也就意味着，在编译某个编译单元时，如果该编译单元会调用到某个内联函数，那么该内联函数的函数定义（即函数体）必须也包含 在该编译单元内。因为编译器使用内联函数体代码替代内联函数调用时，必须知道该内联函数的函数体代码，而且不能通过参考其他编译单元信息来获得这一信息。

如果有多 个编译单元会调用到某同一个内联函数，C++规范要求在这多个编译单元中该内联函数的定义必须是完全一致的，这就是“ODR”（one- definition rule）原则。考虑到代码的可维护性，最好将内联函数的定义放在一个头文件中，用到该内联函数的各个编译单元只需#include 该头文件即可。进一步 考虑，如果该内联函数是一个类的成员函数，这个头文件正好可以是该成员函数所属类的声明所在的头文件。这样看来，类成员内联函数的两种声明可以看成是几乎 一样的，虽然一个是在类外，一个在类内。但是两个都在同一个头文件中，编译器都能在#include 该头文件后直接取得内联函数的函数体代码。讨论完如何 声明一个内联函数，来查看编译器如何内联的。继续上面的例子，假设有个 foo 函数：

```
#include "student.h"

...

void foo()

{

    ...

    Student abc;

    abc.SetAge(12);

    cout << abc.GetAge();

    ...

}
```

foo 函数进入 foo 函数时，从其栈帧中开辟了放置 abc 对象的空间。进入函数体后，首先对该处空间执行 Student 的默认构造函数构造 abc 对象。然后将常数 12 压 栈，调用 abc 的 SetAge 函数（开辟 SetAge 函数自己的栈帧，返回时回退销毁此栈帧）。紧跟着执行 abc 的 GetAge 函数，并将返回值压栈。最 后调用 cout 的<<操作符操作压栈的结果，即输出。

内联后大致如下:

```
#include "student.h"

...

void foo()

{

    ...

    Student abc;

    {
```

```
        abc.age = 12;
    }
    int tmp = abc.age;
    cout << tmp;
    ...
}
```

这时，函数调用时的参数压栈、栈帧开辟与销毁等操作不再需要，而且在结合这些代码后，编译器能进一步优化为如下结果：

```
#include "student.h"
...
void foo()
{
    ...
    cout << 12;
    ...
}
```

这显然是最好的 优化结果；相反，考虑原始版本。如果 SetAge/GetAge 没有被内联，因为非内联函数一般不会在头文件中定义，这两个函数可能在这个编译单元之外的 其他编译单元中定义。即 foo 函数所在编译单元看不到 SetAge/GetAge，不知道函数体代码信息，那么编译器传入 12 给 SetAge，然后用 GetAge 输出。在这一过程中，编译器不能确信最后 GetAge 的输出。因为编译这个编译单元时，不知道这两个函数的函数体代码，因而也就不能做出最终 版本的优化。

从上述分析中，可以看到使用内联函数至少有如下两个优点。

（1）减少因为函数调用引起开销，主要是参数压栈、栈帧开辟与回收，以及寄存器保存与恢复等。

（2）内联后编译器在处理调用内联函数的函数（如上例中的 foo()函数）时，因为可供分析的代码更多，因此它能做的优化更深入彻底。前一条优点对于开发人员来说往往更显而易见一些，但往往这条优点对最终代码的优化可能贡献更大。

这时，有必要简单介绍函数调用时都需要执行哪些操作，这样可以帮助分析一些函数调用相关的问题。假设下面代码：

```
void foo()

{

    ...

    i = func(a, b, c);                    ①

    ...                                   ②

}
```

调用者（这里是 foo）在调用前需要执行如下操作。

（1）参数压栈：这里是 a、b 和 c。压栈时一般都是按照逆序，因此是 c->b->c。如果 a、b 和 c 有对象，则需要先进行拷贝构造（前面章节已经讨论）。

（2）保存返回地址：即函数调用结束返回后接着执行的语句的地址，这里是②处语句的地址。

（3）保存维护 foo 函数栈帧信息的寄存器内容：如 SP（堆栈指针）和 FP（栈帧指针）等。到底保存哪些寄存器与平台相关，但是每个平台肯定都会有对应的寄存器。

（4）保 存一些通用寄存器的内容：因为有些通用寄存器会被所有函数用到，所以在 foo 调用 func 之前，这些寄存器可能已经放置了对 foo 有用的信息。这些寄存器 在进入 func 函数体内执行时可能会被 func 用到，从而被覆写。因此 foo 在调用 func 前保存一份这些通用寄存器的内容，这样在 func 返回后可以恢 复它们。

接着调用 func 函数，它首先通过移动栈指针来分配所有在其内部声明的局部变量所需的空间，然后执行其函数体内的代码等。

最后当 func 执行完毕，函数返回时，foo 函数还需要执行如下善后处理。

（1）恢复通用寄存器的值。

（2）恢复保存 foo 函数栈帧信息的那些寄存器的值。

（3）通过移动栈指针，销毁 func 函数的栈帧，

（4）将保存的返回地址出栈，并赋给 IP 寄存器。

（5）通过移动栈指针，回收传给 func 函数的参数所占用的空间。

在前面章节中已经讨论，如果传入参数和返回值为对象时，还会涉及对象的构造与析构，函数调用的开销就会更大。尤其是当传入对象和返回对象是复杂的大对象时，更是如此。

因为函数调用的准备与善后工作最终都是由机器指令完成的，假设一个函数之前的准备工作与之后的善后工作的指令所需的空间为 SS，执行这些代码所需的时间为 TS，现在可以更细致地从空间与时间两个方面来分析内联的效果。

（1）在 空间上，一般印象是不采用内联，被调用函数的代码只有一份，调用它的地方使用 call 语句引用即可。而采用内联后，该函数的代码在所有调用其处都有一份拷 贝，因此最后总的代码大小比采用内联前要大。但事实不总是这样的，如果一个函数 a 的体代码大小为 AS，假设 a 函数在整个程序中被调用了 n 次，不采用内联 时，对 a 的调用只有准备工作与善后工作两处会增加最后的代码量开销，即 a 函数相关的代码大小为：n * SS + AS。采用内联后，在各处调用点都需要将其函数体代码展开，即 a 函数相关的代码大小为 n * AS。这样比较二者的大小，即比较(n * SS + AS)与(n*AS)的大小。考虑到 n 一般次数很多时，可以简化成比较 SS 与 AS 的大小。这样可以得出大致结论，如果被内联函数自己的函数体代码量比因为 函数调用的准备与善后工作引入的代码量大，内联后程序的代码量会变大；相反，当被内联函数的函数体代码量比因为函数调用的准备与善后工作引入的代码量小， 内联后程序的代码量会变小。这里还没有考虑内联的后续情况，即编译器可能因为获得的信息更多，从而对调用函数的优化做得更深入和彻底，致使最终的代码量变 得更小。

（2）在 时间上，一般而言，每处调用都不再需要做函数调用的准备与善后工作。另外内联后，编译器在做优化时，看到的是调用函数与被调用函数连成的一大块代码。即获 得的代码信息更多，此时它对调用函数的优化可以做得更好。最后还有一个很重要的因素，即内

联后调用函数体内需要执行的代码是相邻的，其执行的代码都在同一 个页面或连续的页面中。如果没有内联，执行到被调用函数时，需要跳到包含被调用函数的内存页面中执行，而被调用函数所属的页面极有可能当时不在物理内存 中。这意味着，内联后可以降低“缺页”的几率，知道减少“缺页”次数的效果远比减少一些代码量执行的效果。另外即使被调用函数所在页面可能也在内存中，但 是因为与调用函数在空间上相隔甚远，所以可能会引起“cache miss”，从而降低执行速度。因此总的来说，内联后程序的执行时间会比没有内联要少。即程序的速度更快，这也是因为内联后代码的空间 “locality”特性提高了。但正如上面分析空间影响时提到的，当 AS 远大于 SS，且 n 非常大时，最终程序的大小会比没有内联时要大很多。代码量大意 味着用来存放代码的内存页也会更多，这样因为执行代码而引起的“缺页”也会相应增多。如果这样，最终程序的执行时间可能会因为大量的“缺页”而变得更多， 即程序的速度变慢。这也是为什么很多编译器对于函数体代码很多的函数，会拒绝对其进行内联的请求。即忽略“inline”关键字，而对如同普通函数那样编 译。

综合上面的分析，在采用内联时需要内联函数的特征。比如该函数自己的函数体代码量，以及程序执行时可能被调用的次数等。当然，判断内联效果的最终和最有效的方法还是对程序的大小和执行时间进行实际测量，然后根据测量结果来决定是否应该采用内联，以及对哪些函数进行内联。

如下根据内联的本质来讨论与其相关的一些其他特点。

如前所 述，因为调用内联函数的编译单元必须有内联函数的函数体代码信息。又因为 ODR 规则和考虑到代码的可维护性，所以一般将内联函数的定义放在一个头文件中， 然后在每个调用该内联函数的编译单元中#include 该头文件。现在考虑这种情况，即在一个大型程序中，某个内联函数因为非常通用，而被大多数编译单元 用到对该内联函数的一个修改，就会引起所有用到它的编译单元的重新编译。对于一个真正的大型程序，重新编译大部分编译单元往往意味着大量的编译时间。因此 内联最好在开发的后期引入，以避免可能不必要的大量编译时间的浪费。

再考虑这 种情况，如果某开发小组在开发中用到了第三方提供的程序库，而这些程序库中包含一些内联函数。因为该开发小组的代码中在用到第三方提供的内联函数处，都是 将该内联函数的函数体代码拷贝到调用处，即该开发小组的代码中包含了第三方提供代码的“实现”。假设这个第三方单位在下一个版本中修改了某些内联函数的定 义，那么虽然这个第三方单位并没有修改任何函数的对外接口，而只是修改了实现，该开发小组要想利用这个新的版本，仍然需要重新编译。考虑到可能该开发小组 的程序已经发布，那么这种重新编译的成本会相当高；相反，如果没有内联，并且仍然只是修改实现，那么该开发小组不必重新编译即可利用新的版本。

因为内联的本质就是用函数体代码代替对该函数的调用，所以考虑递归函数，如：

```
[inline] int foo(int n)
{
    ...
    return foo(n-1);
}
```

如果编译器编译某个调用此函数的编译单元，如：

```
void func()
{
    ...
    int m = foo(n);
    ...
}
```

考虑如下两种情况。

（1）如果在编译该编译单元且调用 foo 时，提供的参数 n 不能知道其实际值，则编译器无法知道对 foo 函数体进行多少次代替。在这种情况下，编译器会拒绝对 foo 函数进行内联。

（2）如果在编译该编译单元且调用 foo 时，提供的参数 n 能够知道其实际值，则编译器可能会视 n 值的大小来决定是否对 foo 函数进行内联。因为如果 n 很大，内联展开可能会使最终程序的大小变得很大。

如前所 述，因为内联函数是编译期行为，而虚拟函数是执行期行为，因此编译器一般会拒绝对虚拟函数进行内联的请求。但是事情总有例外，内联函数的本质是编译器编译 调用某函数时，将其函数体代码代替 call 调用，即内联的条件是编译器能够知道该处函数调用的函数体。而虚拟函数不能够被内联，也是因为在编译时一般来说 编译器无法知道该虚拟函数

到底是哪一个版本，即无法确定其函数体。但是在两种情况下，编译器是能够知道虚拟函数调用的真实版本的，因此虚拟函数可以被内 联。

其一是通过对象，而不是指向对象的指针或者对象的引用调用虚拟函数，这时编译器在编译期就已经知道对象的确切类型。因此会直接调用确定的某虚拟函数实现版本，而不会产生“动态绑定”行为的代码。

其二是虽 然是通过对象指针或者对象引用调用虚拟函数，但是编译时编译器能知道该指针或引用对应到的对象的确切类型。比如在产生的新对象时做的指针赋值或引用初始 化，发生在于通过该指针或引用调用虚拟函数同一个编译单元并且二者之间该指针没有被改变赋值使其指向到其他不能确切知道类型的对象（因为引用不能修改绑 定，因此无此之虞）。此时编译器也不会产生动态绑定的代码，而是直接调用该确定类型的虚拟函数实现版本。

在这两种情况下，编译器能够将此虚拟函数内联化，如：

```
inline virtual int x::y (char* a)

{

   ...

}

void z (char* b)

{

   x_base* x_pointer = new x(some_arguments_maybe);

   x x_instance(maybe_some_more_arguments);

   x_pointer->y(b);

   x_instance.y(b);
```

当然在实际开发中，通过这两种方式调用虚拟函数时应该非常少，因为虚拟函数的语义是“通过基类指针或引用调用，到真正运行时才决定调用哪个版本”。

从上面的 分析中已经看到，编译器并不总是尊重“inline”关键字。即使某个函数用“inline”关键字修饰，并不能够保证该函数在编译时真正被内联处理。因 此与 register 关键字性质类似，inline 仅仅是给编译器的一个“建议”，编译器完全可以视实际情况而忽略之。

另外从内 联，即用函数体代码替代对该函数的调用这一本质看，它与 C 语言中的函数宏（macro）极其相似，但是它们之间也有本质的区别。即内联是编译期行为，宏是 预处理期行为，其替代展开由预处理器来做。也就是说编译器看不到宏，更不可能处理宏。另外宏的参数在其宏体内出现两次或两次以上时经常会产生副作用，尤其 是当在宏体内对参数进行++或--操作时，而内联不会。还有，预处理器不会也不能对宏的参数进行类型检查。而内联因为是编译器处理的，因此会对内联函数的 参数进行类型检查，这对于写出正确且鲁棒的程序，是一个很大的优势。最后，宏肯定会被展开，而用 inline 关键字修饰的函数不一定会被内联展开。

最后顺带提及，一个程序的惟一入口 main()函数肯定不会被内联化。另外，编译器合成的默认构造函数、拷贝构造函数、析构函数，以及赋值运算符一般都会被内联化。

inline 函数

我们看下面的函数，函数体中只有一行语句：

```
double Average(double total, int number){
  return total/number;
  }
```

定义这么简单的函数有必要吗？实际上，它还是有一些优点的：第一，它使程序更可读；第二，它使这段代码可以重复使用。但是，它也有缺点：当它被频繁地调用的时候，由于调用函数的开销，会对应用程序的性能(时间＋空间效率，这儿特指时间)有损失。例如，Average 在一个循环语句中重复调用几千次，会降低程序的执行效率。

那么，有办法避免函数调用的开销吗？对于上面的函数，我么可以把它定义为内联函数的形式：

```
inline double Average(double total, int number){
  return total/number;
}
```

函数的引入可以减少程序的目标代码，实现程序代码的共享。

函数调用需要时间和空间开销，调用函数实际上将程序执行流程转移到被调函数中，被调函数的代码执行完后，再返回到调用的地方。这种调用操作要求调用前保护好现场并记忆执行的地址，返回后恢复现场，并按原来保存的地址继续执行。对于较长的函数这种开销可以忽略不计，但对于一些函数体代码很短，又被频繁调用的函数，就不能忽视这种开销。引入内联函数正是为了解决这个问题，提高程序的运行效率。

在程序编译时，编译器将程序中出现的内联函数的调用表达式用内联函数的函数体来进行替换。由于在编译时将内联函数体中的代码替代到程序中，因此会增加目标程序代码量，

进而增加空间开销，而在时间开销上不象函数调用时那么大，可见它是以目标代码的增加为代价来换取时间的节省。

◆总结：inline 函数是提高运行时间效率，但却增加了空间开销。
即 inline 函数目的是：为了提高函数的执行效率(速度)。
非内联函数调用有栈内存创建和释放的开销
在 C 中可以用宏代码提高执行效率，宏代码不是函数但使用起来像函数，编译器用复制宏代码的方式取代函数调用，省去了参数压栈、生成汇编语言的 CALL 调用、返回参数、执行 return 等过程，从而提高速度。

◆使用宏的缺点：（1）容易出错(预处理器在复制宏代码时常常产生意想不到的边际效应)
例如：#define MAX(a,b)　(a) > (b) ? (a) : (b)
语句 result = MAX(i,j) + 2 却被扩展为 result = (i)>(j)?(i):(j)+2;
但意却为 result = ((i)>(j)?(i):(j)) + 2;
（2）不可调试
（3）无法操作类的私有数据成员

C++函数内联机制既具备宏代码的效率，又增加了安全性，且可自由操作类的数据成员。

关键字 inline 必须与函数定义体放在一起才能使函数真正内联，仅把 inline 放在函数声明的前面不起任何作用。因为 inlin 是一种用于实现的关键字，不是一种用于声明的关键字。

许多书籍把内联函数的声明、定义体前都加了 inline 关键字，但声明前不应该加(加不加不会影响函数功能)，因为声明与定义不可混为一谈。

★声明、定义和语句
声明：就是在向系统介绍名字（一个名字是一块内存块的别名），只是告诉编译器这个名字值的类型及宣告该名字的存在性，仅此而已。
定义：则是分配存储空间，即具有了存储类型。
语句：程序的基本组成部分，分可执行语句(定义是)和不可执行语句(声明是)。
在正式编写程序语句前定义的一些全局变量或局部变量，在 C 中为声明，C++中为定义。
例如：int　a;//在标 C 中为声明，是不可执行语句；在 C++中为定义，是可执行语句
extern int a;//为声明，是不可执行语句　CWinApp　curApp;//对象定义是可执行语句

◆使用内联函数时应注意以下几个问题：
（1） 在一个文件中定义的内联函数不能在另一个文件中使用。它们通常放在头文件中共享。
（2） 内联函数应该简洁，只有几个语句，如果语句较多，不适合于定义为内联函数。
（3） 内联函数体中，不能有循环语句、if 语句或 switch 语句，否则，函数定义时即使有 inline 关键字，编译器也会把该函数作为非内联函数处理。
（4） 内联函数要在函数被调用之前声明。
例如：
#include <iostream.h>

```
int increment(int i);
inline int increment(int i){
  i++; return i;
}
void main(void){  ……
}
```
如果我们修改一下程序，将内联函数的定义移到 main()之后：
```
    #include <iostream.h>
int increment(int i);
void main(void){  ……
}
//内联函数定义放在 main()函数之后
inline int increment(int i){
  i++; return i;
}
```
内联函数在调用之后才定义，这段程序在编译的时候编译器不会直接把它替换到 main 中。也就是说实际上"increment(int i)"只是作为一个普通函数被调用，并不具有内联函数的性质，无法提高运行效率。