# 函数调用关系

# 内存处理基本工具（ <stl\_uninitialized.h> ）

1. uninitialized\_copy 函数簇：

作用：[\_\_first , \_\_last)的值，复制给 \_\_result；

uninitialized\_copy ： 底层接口（功能实现）

1. inline \_ForwardIter \_\_uninitialized\_copy\_aux(\_InputIter \_\_first, \_InputIter \_\_last, \_ForwardIter \_\_result, \_\_true\_type) ;
2. \_ForwardIter \_\_uninitialized\_copy\_aux(\_InputIter \_\_first, \_InputIter \_\_last, \_ForwardIter \_\_result, \_\_false\_type);

uninitialized\_copy ： 中间层接口（判断型别是否为 \_Is\_POD ： 类或者基本类型）

1. inline \_ForwardIter \_\_uninitialized\_copy (\_InputIter \_\_first , \_InputIter \_\_last,\_ForwardIter \_\_result, \_Tp\*);

uninitialized\_copy ： 上层接口（传递 迭代器所指对象的类型， 原生指针）

1. inline \_ForwardIter uninitialized\_copy(\_InputIter \_\_first, \_InputIter \_\_last, \_ForwardIter \_\_result)
2. inline char\* uninitialized\_copy(const char\* \_\_first, const char\* \_\_last, char\* \_\_result)
3. inline wchar\_t\* uninitialized\_copy(const wchar\_t\* \_\_first, const wchar\_t\* \_\_last,wchar\_t\* \_\_result
4. uninitialized\_copy\_n 函数簇

作用：[ \_\_first , \_\_first+\_\_count )的值，复制给 \_\_result ；

uninitialized\_copy\_n ： 底层接口（功能实现）

1. pair<\_InputIter, \_ForwardIter> \_\_uninitialized\_copy\_n(\_InputIter \_\_first, \_Size \_\_count, \_ForwardIter \_\_result, input\_iterator\_tag);
2. inline pair<\_RandomAccessIter, \_ForwardIter> \_\_uninitialized\_copy\_n(\_RandomAccessIter \_\_first, \_Size \_\_count, \_ForwardIter \_\_result, random\_access\_iterator\_tag);

uninitialized\_copy\_n ： 上层接口（根据迭代器类型，选择底层接口）

1. inline pair<\_InputIter, \_ForwardIter> uninitialized\_copy\_n(\_InputIter \_\_first, \_Size \_\_count, \_ForwardIter \_\_result);
2. uninitialized\_fill 函数簇

作用：将[\_\_first , \_\_last )的空间，全部填充成\_\_x；

uninitialized\_fill ： 底层接口（功能实现）

1. inline void \_\_uninitialized\_fill\_aux( \_ForwardIter \_\_first, \_ForwardIter \_\_last , const \_Tp& \_\_x, \_\_true\_type);
2. void \_\_uninitialized\_fill\_aux(\_ForwardIter \_\_first, \_ForwardIter \_\_last, const \_Tp& \_\_x, \_\_false\_type);

uninitialized\_fill ：中间层接口（判断型别是否为 \_Is\_POD ： 类或者基本类型）

1. inline void \_\_uninitialized\_fill(\_ForwardIter \_\_first , \_ForwardIter \_\_last , const \_Tp& \_\_x, \_Tp1\*);

uninitialized\_fill ：上层接口（传递 迭代器的所指对象的类型（ value\_type ） ）

1. inline void uninitialized\_fill (\_ForwardIter \_\_first , \_ForwardIter \_\_last , const \_Tp& \_\_x);
2. \_\_uninitialized\_copy\_copy 函数

作用：

[\_\_first , \_\_last)的值 ，复制给[ result , result + (last1 - first1) ) ；

此时 result 在 result + (last1 - first1) 的位置

[first2, last2) 的值，复制给 [ result , result + (last2 - first2) )

1. inline \_ForwardIter \_\_uninitialized\_copy\_copy(\_InputIter1 \_\_first1 , \_InputIter1 \_\_last1 , \_InputIter2 \_\_first2, \_InputIter2 \_\_last2 , \_ForwardIter \_\_result)
2. \_\_uninitialized\_fill\_copy 函数

作用：

[\_\_result, \_\_mid)区间 ，全部填充为\_\_x ；

[\_\_first , \_\_last)的值 ，复制给 [ mid , mid+(\_\_last - \_\_first) ]；

1. inline \_ForwardIter \_\_uninitialized\_fill\_copy(\_ForwardIter \_\_result, \_ForwardIter \_\_mid , const \_Tp& \_\_x , \_InputIter \_\_first , \_InputIter \_\_last)
2. \_\_uninitialized\_copy\_fill 函数

作用：

[\_\_first1, \_\_last1)区间 ，复制给 [\_\_first2, \_\_first2+(\_\_last1 - \_\_first1) ] ；

[ first2 + (last1 - first1), last2 )区间， 填充为 x ；

1. inline void \_\_uninitialized\_copy\_fill(\_InputIter \_\_first1, \_InputIter \_\_last1,\_ForwardIter \_\_first2, \_ForwardIter \_\_last2 , const \_Tp& \_\_x)

# 对象的构造与析构（ < stl\_construct.h> ）

1. construct ：构造对象

construct：底层接口（ 定位new表达式：new \_\_T1() 使用的是\_\_p指向的地址（不重新分配空间） ）

1. inline void \_Construct(\_T1\* \_\_p, const \_T2& \_\_value);
2. inline void \_Construct(\_T1\* \_\_p);

construct：旧接口（直接调用，底层接口）

1. inline void construct(\_T1\* \_\_p);
2. inline void construct(\_T1\* \_\_p, const \_T2& \_\_value);
3. destory ：析构对象

destory：底层接口（调用对象的析构函数）

* + 1. inline void \_Destroy(\_Tp\* \_\_pointer);

destory：次中间层接口（遍历迭代器区间内容，释放迭代器所指对象的空间）

* + 1. void \_\_destroy\_aux(\_ForwardIterator \_\_first, \_ForwardIterator \_\_last, \_\_false\_type) ;
    2. inline void \_\_destroy\_aux(\_ForwardIterator, \_ForwardIterator, \_\_true\_type);

destory：中间层接口（萃取“型别”，获取是否有析构函数）

* + 1. inline void \_\_destroy(\_ForwardIterator \_\_first, \_ForwardIterator \_\_last, \_Tp\*);

destory：上层接口（萃取“迭代器”，获取型别）

* + 1. inline void \_Destroy(\_ForwardIterator \_\_first, \_ForwardIterator \_\_last);
    2. inline void \_Destroy(char\*, char\*);
    3. inline void \_Destroy(int\*, int\*);
    4. inline void \_Destroy(long\*, long\*);
    5. inline void \_Destroy(float\*, float\*);
    6. inline void \_Destroy(double\*, double\*);
    7. inline void \_Destroy(wchar\_t\*, wchar\_t\*);

destroy：旧接口

* + 1. inline void destroy(\_Tp\* \_\_pointer);
    2. inline void destroy(\_ForwardIterator \_\_first, \_ForwardIterator \_\_last);

# 空间配置器（ <stl\_alloc.h> ）

1. 空间配置器属性:
2. allocator::value\_type
3. allocator::pointer
4. allocator::const\_pointer
5. allocator::reference
6. allocator::const\_reference
7. allocator::size\_type
8. allocator::difference\_type
9. allocator::rebind

一个嵌套的class template ， class rebind<U> 拥有唯一的成员other ， 那是一个typedef，代表allocator<U>

1. 空间配置器的接口：
2. allocator::allocator();
3. allocator::allocator(const allocator&)
4. template<class U> allocator::allocator(const allocator<U>&)
5. allocator::~allocator()
6. pointer allocator::address(reference x) ;

//返回某个对象的地址, a.address(x) 等同于 &x

1. const\_pointer allocator::address(const reference x) const

//返回某个const对象的地址

1. pointer allocator::allocate(size\_type n , const void\* = 0)

//配置空间，足以存储n个T对象；

//第二个参数是个提示，实现上可能会利用它来增加区域性，或完全忽略之；

1. pointer allocator::deallocate(pointer p , size\_type n )

//归还先前配置的空间

1. size\_type allocator::max\_size() const

//返回可成功配置的最大量

1. void allocator::construct(pointer p , const T& x)

//等同于 new ( (const void \*) p ) T(x);

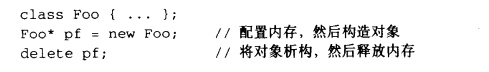
1. void allocator::destroy(pointer p)

//等同于 p->~T();

1. SGI标准空间配置器（std::allocator）

SGI从未使用，主要是因为效率不佳，只是把C++ ::operator new 和 ::operator delete做了一层简单包装；

1. SGI特殊空间配置器（std::alloc）



new操作内含两阶段操作：

* 1. ::operator new 配置内存 ；
  2. ::Foo::Foo() 构造对象；

delete操作内涵两段操作：

* 1. Foo::~Foo() 析构对象；
  2. ::operator delete 释放内存；

1. 为精密分工，STL alloc 决定将两段操作区分开来:

内存配置操作 -- alloc::allocate()；

内存释放操作 -- alloc::deallocate();

对象构造操作 -- ::construct(); 详情见< stl\_construct.h>

对象析构操作 -- ::destroy(); 详情见< stl\_construct.h>

1. 对象构造前的空间配置及对象析构后的空间释放由<stl\_alloc.h>负责，SGI对此设计哲学如下：
   * + 1. 向system heap要求空间；
       2. 考虑多线程状态；
       3. 考虑内存不足时应变策略；
       4. 考虑过多“小型区块”可能造成的内存碎片（fragment）问题；

* STL如何应对小型区块可能造成的内存碎片问题？

SGI设计了双层级配置器：

第一级配置器：直接使用malloc 和 free ;

第二级配置器：采用不同策略；

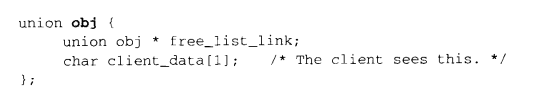
策略（次层配置策略）如下：

1. 区块（>128bytes）时，移交给“一级配置器”处理；
2. 区块（<=128bytes）时，以 "内存池(memory pool)" 方式管理；

具体实施方式：

* + - * 1. 每次配置一大块内存，并维护对应之自由链表（free-list）；
        2. 下次若再有相同大小的内存需求，就直接从free-list中拨出。
        3. 如果客端释还小额区块，就由配置器回收到free-list中；

为了方便管理，SGI第二级配置器主动将任何小额区块的内存需求量上调至8的倍数；（例如：客端要求30 bytes , 就自动调整为32 bytes）,并维护16个free-list( union obj free\_list[16] )，各自管理大小分别为：8，16，24，32，40，48，56，64，72，80，88，96，104，112，120，128byte的小额区块；



union使得SGI STL不会为了维护链表所必须的指针而造成另一种内存浪费；

* 如何将内存需求量上调至N的倍数算法(N为整数)？

static size\_t \_S\_round\_up(size\_t \_\_bytes)

{ return ( ( (\_\_bytes) + N - 1 ) & ~( N - 1 ) ); }

* 如何查找“上调至N的数据”存储空间的位置？

static size\_t \_S\_freelist\_index(size\_t \_\_bytes)

{

return (((\_\_bytes) + (size\_t)\_N - 1)/(size\_t) N - 1);

}

* STL如何考虑内存不足时的应变策略？

第一级配置器：实现类似C++ new handler的机制 ，因为第一级配置器并非使用::operator new来配置内存，所以不能直接运用C++ new handler机制，该机制成为out-of-memory处理机制；

New handler机制：要求系统在内存配置需求无法被满足时，在抛出std::bad\_alloc异常状态之前，调用一个客端指定的函数（处理例程）；new handler解决内存不足做法有特定模式；

SGI第一级配置器的allocate() 和 realloc()都是在调用malloc() 和 realloc()不成功后，改调用oom\_malloc()和oom\_realloc()。后两者内部都有内循环，不断调用“内存不足处理例程”

设计“New handle内存不足处理例程”是客端的责任，设定“New handle内存不足处理例程”也是客端的责任；

1. STL整个设计开放“第几级配置器”，取决于\_\_USE\_MALLOC宏；

#if \_\_USE\_MALLOC //第一级配置器

typedef \_\_malloc\_alloc\_template<0> malloc\_alloc;

typedef malloc\_alloc alloc;

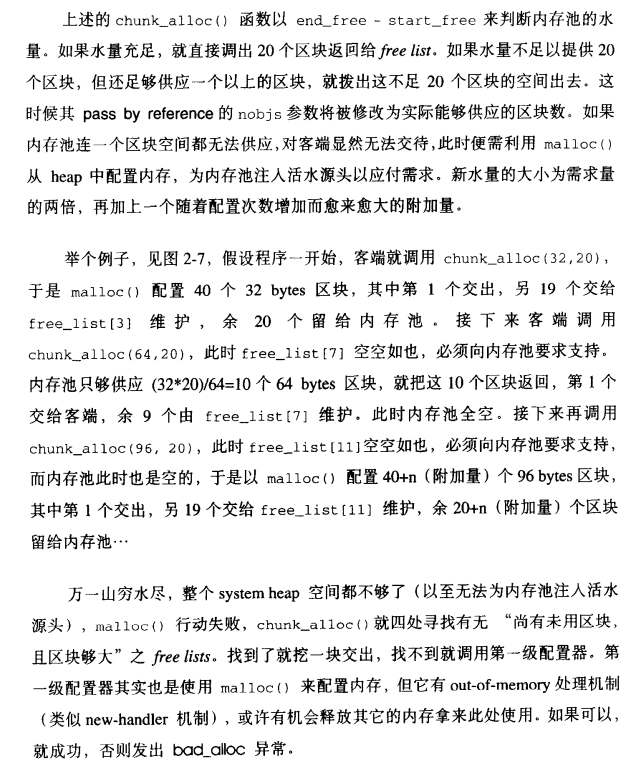
#else //第二级配置器

Typedef \_\_default\_alloc\_template<\_\_NODE\_ALLOCATOR\_THREADS,0> alloc;

#endif

1. STL包装了一个接口，使配置器接口能够符合STL规格class simple\_alloc ,STL容器全部使用这个接口；
2. 空间配置函数(allocate) / 空间释放函数（deallocate） / 重新填充free\_list（refill）
3. 内存池（memory pool）

从内存池取空间给free list 使用，是chunk\_alloc()的工作；



1. static void (\* \_\_set\_malloc\_handler( void (\*\_\_f)() ) ) () ，该函数如何调用？

返回值（类型）：void (\*)()

函数名：\_\_set\_malloc\_handler

参数: void (\*\_\_f)() 结合方式： （） 优先级大于 \* ， 所以需要：（\*\_\_f）

此处谨记：

函数定义：类型 函数名(形参列表)

变量定义：类型 变量名；

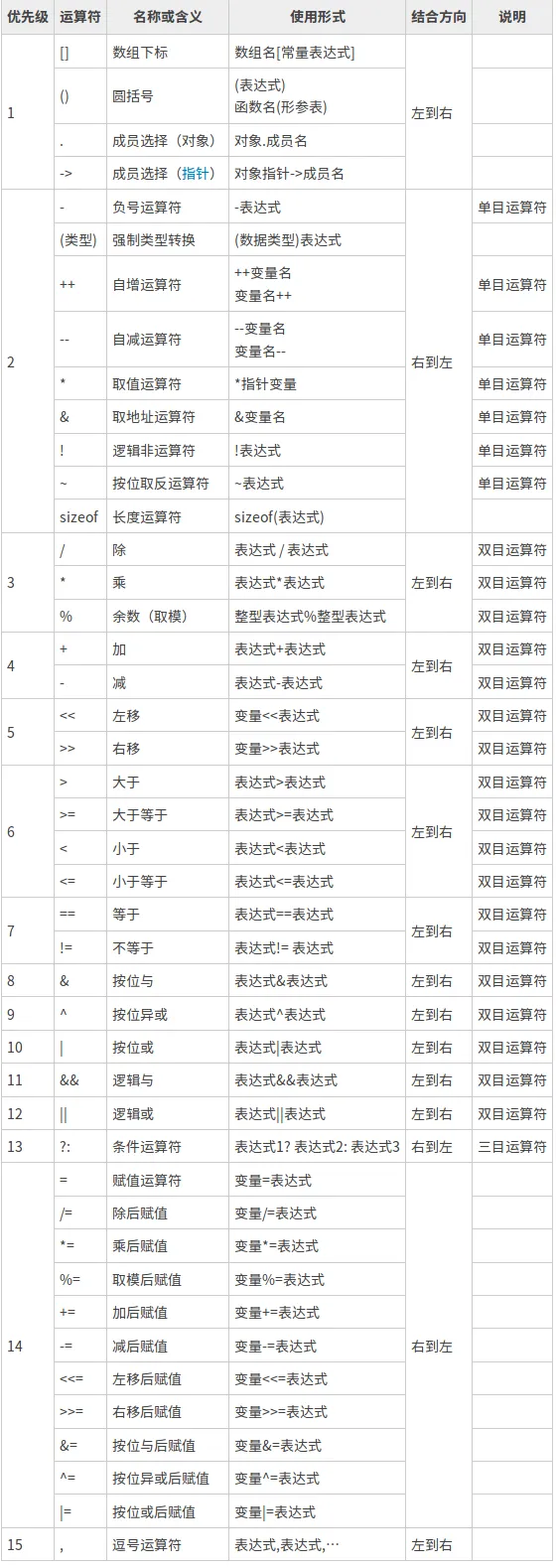
调用：

Void (\*foo)();

Void (\*bar)();

foo = \_\_set\_malloc\_handler (bar);

运算符优先级：



1. void \*realloc( void \* p , size\_t n); -- 指针函数

返回值：void \*;

参数1：void \*p 原有的空间地址；

参数2：size\_t n 重新申请的空间大小；

realloc执行流程:

* + - 1. realloc在原空间后面申请一段空间
      2. 申请成功：函数执行完成；

申请失败：调用malloc重新申请一块空间；

* + - 1. 重新申请空间：

申请成功：将原空间数据拷贝到新空间中，并释放原空间，并返回新数组首地址；

申请失败：不会释放掉原数组，并返回一个NULL；

1. C预处理器详解

* C预处理器命名规则：以“#”开头，第一个必须为“非空字符”，出现在第一列；
* 预处理指令

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

* 预定义宏（ANSI C 定义了许多宏，可以使用但不能直接修改这些宏）：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

* 预处理器运算符
  + 1. 宏延续运算符（\）：当宏定义需要换行时，可以使用宏延续运算符（因为宏定义只能是单行）；
    2. 字符串常量化运算符（#）：当需要把一个宏的参数转换成字符串常量时，使用字符串常量化运算符（#）；

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

* + 1. 标记粘贴运算符（##）：允许在宏定义中，两个独立的标记合并为一个标记

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

* + 1. defined() 运算符：

用来确定一个标识符是否已经使用#define定义过；

标识符已定义：则值为非0；

标识符未定义：则值为0；

* + 1. 参数化宏：

图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

1. C共用体

共用体特点：

1. 相同的内存位置存储不同的数据类型；
2. 可以带有多个成员，但同一时间只能有一个成员带有值；
3. Volatitle关键字

作用：告诉编译器它后面所定义的变量随时都有可能改变；

因此编译后，程序每次存储或读取这个变量时，都会直接从变量地址中读取数据，

如果没有volatile 编译器可能优化读取和存储，可能暂时使用寄存器中的值；