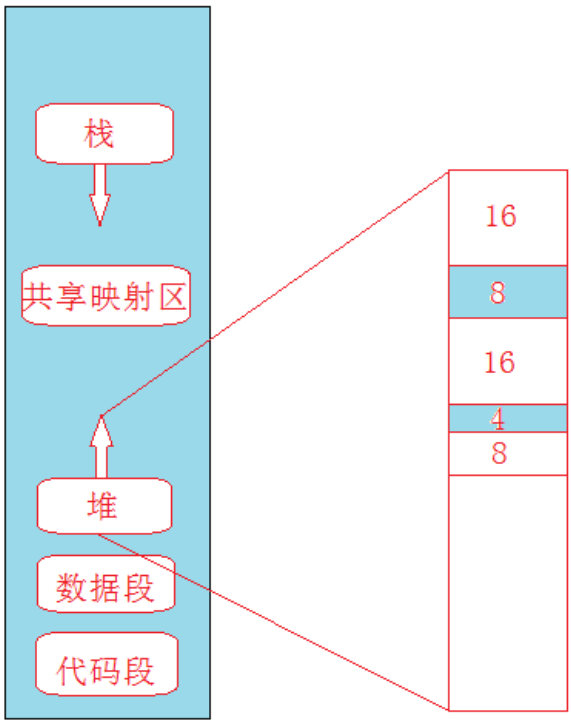
# 3-空间配置器-allocator

----------比特科技整理----------

- 本节目标

1. STL为什么需要空间配置器？
2. 还有其他场景下需要内存池。
3. 空间配置器的缺点
4. 配合使用的一些泛型算法
5. 终极目标--模拟实现一个简洁版的空间配置器

【为什么需要空间配置器？】
1:内存碎片问题（外碎片）




假设系统依次分配了16byte、8byte、16byte、4byte，还剩余8byte未分配。
这时要分配一个24byte的空间，操作系统回收了一个上面的两个16byte，总的剩余空间有40byte，但是却不能分配出一个连续24byte的空间，这就是内存碎片问题。

2：频繁的分配小块内存，效率比较低。

- 【STL空间配置器的框架设计】

**1：设计一级空间配置器和二级空间配置器**

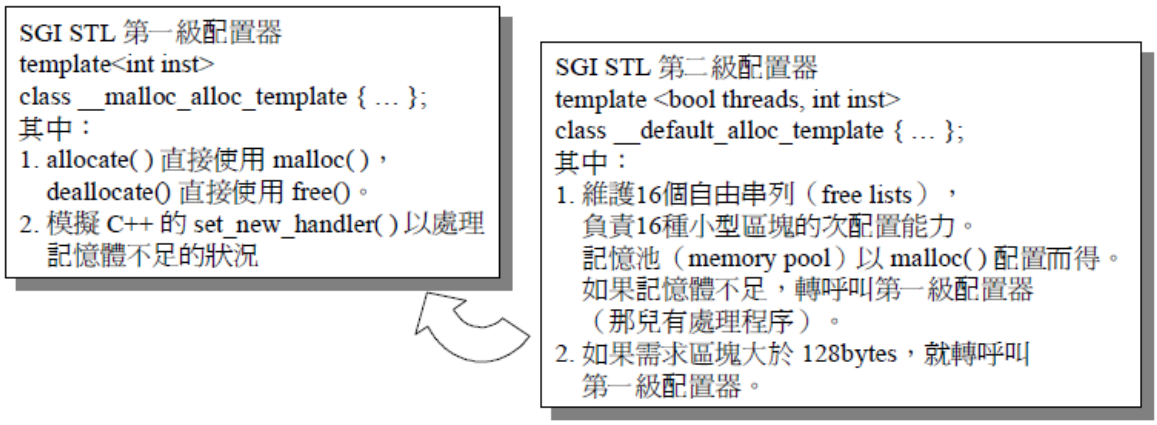



圖 2-2a　第一級配置器與第二級配置器

**2：通过__USE_MALLOC宏配置是否使用二级空间配置器**

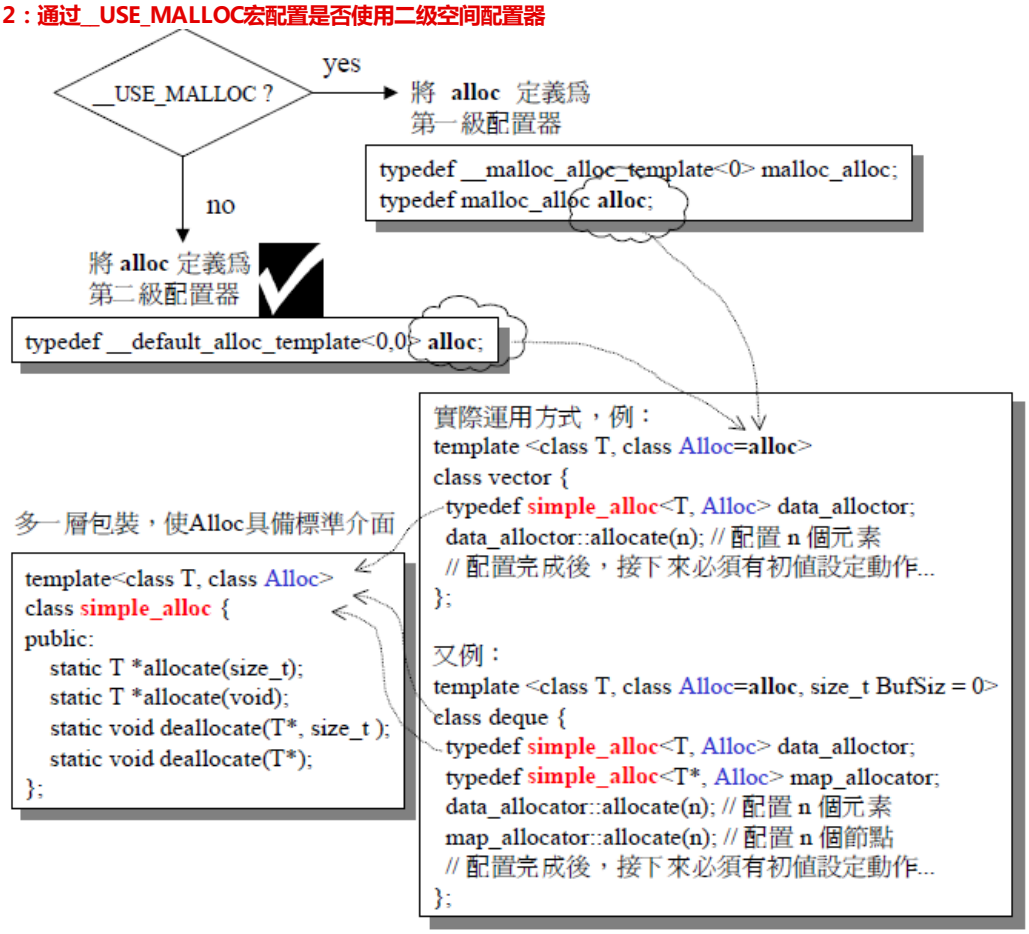



圖 2-2b　第一級配置器與第二級配置器，其包裝介面和運用方式

```
//////////////////////////////////////////////////////////////////////
//一级空间配置器（malloc/realloc/free）

// 内存分配失败以后处理的句柄 handler类型
typedef void(*ALLOC_OOM_FUN)();
template <int inst>
class __MallocAllocTemplate
{
private:
    //static void (* __sMallocAllocOomHandler)();
    static ALLOC_OOM_FUN __sMallocAllocOomHandler;

    static void* OomMalloc(size_t n)
    {
        ALLOC_OOM_FUN handler;
```

```
        void* result ;

        //
        // 1：分配内存成功，则直接返回
        // 2：若分配失败，则检查是否设置处理的 handler，
        // 有则调用以后再分配。不断重复这个过程，直到分配成功为止。
        // 没有设置处理的handler，则直接结束程序。
        //
        for (;;) {
            handler = __sMallocAllocOomHandler ;
            if (0 == handler )
            {
                cerr<<"out of memory" <<endl;
                exit(-1);
            }

            handler();

            result = malloc (n);
            if (result )
                return(result );
        }
    }

    static void *OomRealloc( void* p , size_t n)
    {
        // 同上
        ALLOC_OOM_FUN handler ;
        void* result ;

        for (;;) {
            handler = __sMallocAllocOomHandler ;
            if (0 == handler )
            {
                cerr<<"out of memory" <<endl;
                exit(-1);
            }

            (* handler)();
            result = realloc (p, n);
            if (result ) return( result);
        }
    }
public :
    static void * Allocate( size_t n )
    {
        __TRACE_DEBUG("(n:%u)\n", n);

        void *result = malloc( n);
        if (0 == result ) result = OomMalloc(n );
        return result ;
    }

    static void Deallocate( void *p , size_t /* n */)
    {
        __TRACE_DEBUG("(p:%p)\n", p);

        free(p );
    }

    static void * Reallocate( void *p , size_t /* old_sz */, size_t new_sz )
    {
        void * result = realloc( p, new_sz );
        if (0 == result ) result = OomRealloc(p , new_sz);
        return result ;
    }

    static void (* SetMallocHandler( void (*f) ()))()
    {
        void (* old )() = __sMallocAllocOomHandler ;
        __sMallocAllocOomHandler = f;
        return(old );
    }
```

```
};

// 分配内存失败处理函数的句柄函数指针
template <int inst>
ALLOC_OOM_FUN __MallocAllocTemplate <inst>:: __sMallocAllocOomHandler = 0;
```

### 3：二级空间配置器


```
//////////////////////////////////////////////////////////////////////
// 二级空间配置器
//
template <bool threads, int inst>
class __DefaultAllocTemplate
{
public :
    enum {__ALIGN = 8};                         // 排列基准值（也是排列间隔）
    enum {__MAX_BYTES = 128};                   // 最大值
    enum {__NFREELISTS = __MAX_BYTES/ __ALIGN};    // 排列链大小

    static size_t ROUND_UP( size_t bytes )
    {
        // 对齐
        return ((bytes + __ALIGN - 1) & ~(__ALIGN - 1));
    }

    static size_t FREELIST_INDEX( size_t bytes )
    {
        // bytes == 9
        // bytes == 8
        // bytes == 7
        return ((bytes + __ALIGN - 1)/ __ALIGN - 1);
    }

    union Obj
    {
        union Obj * _freeListLink;    // 指向下一个内存块的指针
        char _clientData [1];    /* The client sees this.*/
    };

    static Obj * volatile _freeList[__NFREELISTS ];     // 自由链表
    static char * _startFree;                           // 内存池水位线开始
    static char * _endFree;                             // 内存池水位线结束
    static size_t _heapSize;                            // 从系统堆分配的总大小

    // 获取大块内存插入到自由链表中
    static void * Refill( size_t n );
    // 从内存池中分配大块内存
    static char * ChunkAlloc( size_t size , int &nobjs );

    static void * Allocate( size_t n );
    static void Deallocate( void *p , size_t n);
    static void * Reallocate( void *p , size_t old_sz, size_t new_sz);
```

```
};

// 初始化全局静态对象
template <bool threads, int inst>
typename __DefaultAllocTemplate< threads, inst>::Obj * volatile __DefaultAllocTemplate <threads, inst>::_freeList
[__DefaultAllocTemplate< threads, inst>::__NFREELISTS ];

template <bool threads, int inst>
char* __DefaultAllocTemplate <threads, inst>::_startFree = 0;
template <bool threads, int inst>
char* __DefaultAllocTemplate <threads, inst>::_endFree = 0;
template <bool threads, int inst>
size_t __DefaultAllocTemplate< threads, inst>::_heapSize = 0;;

template <bool threads, int inst>
void* __DefaultAllocTemplate <threads, inst>::Refill (size_t n)
{
    __TRACE_DEBUG("(n:%u)\n", n);

    //
    // 分配个n bytes的内存
    // 如果不够则能分配多少分配多少
    //
    int nobjs = 20;
    char* chunk = ChunkAlloc( n, nobjs );

    // 如果只分配到一块，则直接这块内存。
    if(nobjs == 1)
        return chunk ;

    Obj* result , *cur;
    size_t index = FREELIST_INDEX( n);
    result = (Obj *)chunk;

    // 把剩余的块链接到自由链表上面
    cur = (Obj *)(chunk+ n);
    _freeList[index ] = cur;
    for(int i = 2; i < nobjs ; ++i)
    {
        cur->_freeListLink = (Obj*)( chunk+n *i);
        cur = cur ->_freeListLink;
    }

    cur->_freeListLink = NULL;
    return result ;
}

template <bool threads, int inst>
char* __DefaultAllocTemplate <threads, inst>::ChunkAlloc (size_t size, int &nobjs)
{
    __TRACE_DEBUG("(size: %u, nobjs: %d)\n", size, nobjs );

    char* result ;
    size_t bytesNeed = size* nobjs;
    size_t bytesLeft = _endFree - _startFree;

    //
    // 1.内存池中的内存足够，bytesLeft>=bytesNeed，则直接从内存池中取。
    // 2.内存池中的内存不足，但是够一个 bytesLeft >= size，则直接取能够取出来。
    // 3.内存池中的内存不足，则从系统堆分配大块内存到内存池中。
    //
    if (bytesLeft >= bytesNeed)
    {
        __TRACE_DEBUG(" 内存池中内存足够分配 %d个对象\n", nobjs);

        result = _startFree ;
        _startFree += bytesNeed ;
    }
    else if (bytesLeft >= size)
    {
        __TRACE_DEBUG(" 内存池中内存不够分配 %d个对象，只能分配%d个对象 \n", nobjs , bytesLeft / size);
        result = _startFree ;
```

```
            nobjs = bytesLeft / size;
            _startFree += nobjs *size;
        }
        else
        {
            // 若内存池中还有小块剩余内存，则将它头插到合适的自由链表
            if (bytesLeft > 0)
            {
                size_t index = FREELIST_INDEX (bytesLeft);
                ((Obj*)_startFree)->_freeListLink = _freeList[index];
                _freeList[index] = (Obj*) _startFree;
                _startFree = NULL ;

                __TRACE_DEBUG("将内存池中剩余的空间，分配给 freeList[%d]\n", index);
            }

            // 从系统堆分配两倍+已分配的 heapSize/8的内存到内存池中
            size_t bytesToGet = 2*bytesNeed + ROUND_UP(_heapSize >>4);
            _startFree = (char *)malloc(bytesToGet);
            __TRACE_DEBUG("内存池空间不足，系统堆分配 %u bytes内存\n", bytesToGet);

            //
            // 【无奈之举】
            // 如果在系统堆中内存分配失败，则尝试到自由链表中更大的节点中分配
            //
            if (_startFree == NULL)
            {
                __TRACE_DEBUG("系统堆已无足够，无奈之下，智能到自由链表中看看 \n");

                for(int i = size; i <= __MAX_BYTES; i+=__ALIGN)
                {
                    Obj* head = _freeList[FREELIST_INDEX (size)];
                    if (head)
                    {
                        _startFree = (char *)head;
                        head = head->_freeListLink;
                        _endFree = _startFree +i;
                        return ChunkAlloc (size, nobjs);
                    }
                }

                //
                // 【山穷水尽，最后一根稻草】
                // 自由链表中也没有分配到内存，则再到一级配置器中分配内存，
                // 一级配置器中可能有设置的处理内存，或许能分配到内存。
                //
                __TRACE_DEBUG("系统堆和自由链表都已无内存，一级配置器做最后一根稻草 \n");
                _startFree = (char*)MallocAlloc ::Allocate(bytesToGet);
            }

            // 从系统堆分配的总字节数。（可用于下次分配时进行调节）
            _heapSize += bytesToGet ;
            _endFree = _startFree + bytesToGet;

            // 递归调用获取内存
            return ChunkAlloc (size, nobjs);
        }

        return result ;
    }

    template <bool threads, int inst>
    void* __DefaultAllocTemplate <threads, inst>::Allocate (size_t n)
    {
        __TRACE_DEBUG("(n: %u)\n", n);

        //
        // 若n > __MAX_BYTES 则直接在一级配置器中获取
        // 否则在二级配置器中获取
        //
        if (n > __MAX_BYTES)
        {
```

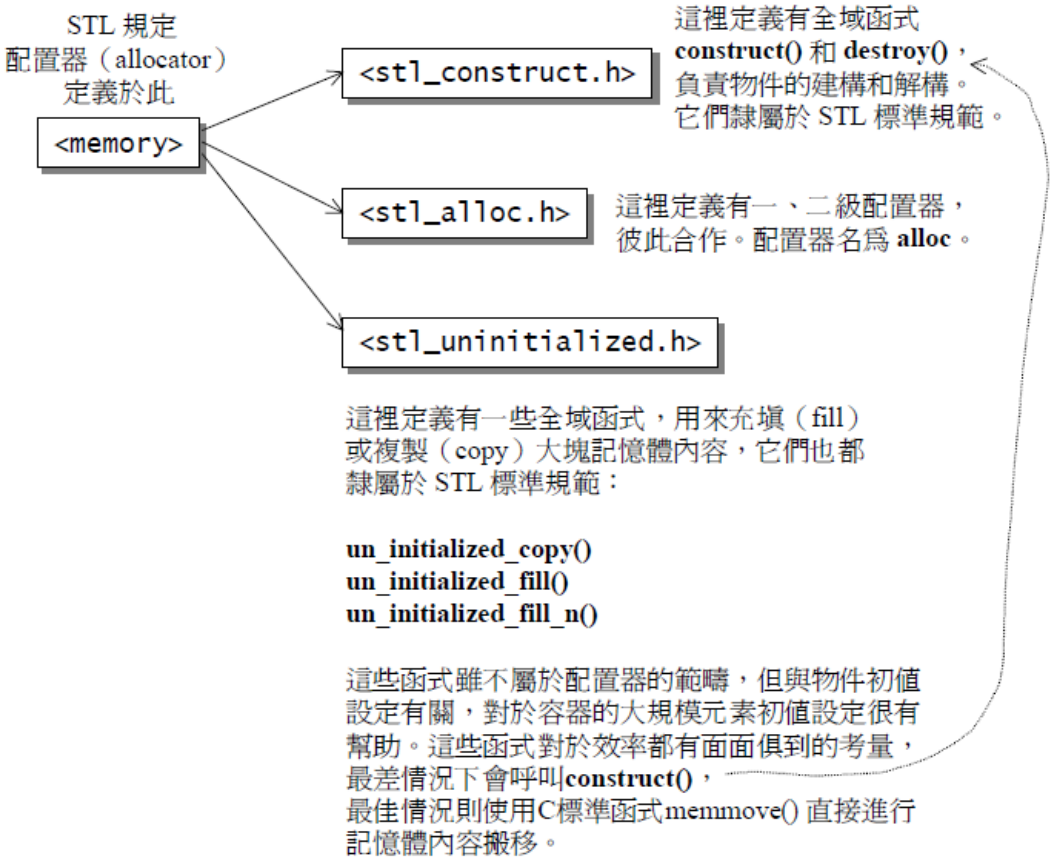
```
        return MallocAlloc::Allocate(n);
    }

    size_t index = FREELIST_INDEX(n);
    void* ret = NULL;

    //
    // 1.如果自由链表中没有内存则通过 Refill进行填充
    // 2.如果自由链表中有则直接返回一个节点块内存
    // ps:多线程环境需要考虑加锁
    //
    Obj* head = _freeList[index];
    if (head == NULL)
    {
        return Refill(ROUND_UP(n));
    }
    else
    {
        __TRACE_DEBUG("自由链表取内存:_freeList[%d]\n", index);

        _freeList[index] = head->_freeListLink;
        return head;
    }
}

template<bool threads, int inst>
void __DefaultAllocTemplate<threads, inst>::Deallocate(void* p, size_t n)
{
    __TRACE_DEBUG("(p:%p, n: %u)\n", p, n);


    //
    // 若n > __MAX_BYTES 则直接归还给一级配置器
    // 否则在放回二级配置器的自由链表
    //
    if (n > __MAX_BYTES)
    {
        MallocAlloc::Deallocate(p, n);
    }
    else
    {
        // ps:多线程环境需要考虑加锁
        size_t index = FREELIST_INDEX(n);

        // 头插回自由链表
        Obj* tmp = (Obj*)p;
        tmp->_freeListLink = _freeList[index];
        _freeList[index] = tmp;
    }
}

template<bool threads, int inst>
void* __DefaultAllocTemplate<threads, inst>::Reallocate(void* p, size_t old_sz, size_t new_sz)
{
    void* result;
    size_t copy_sz;

    if (old_sz > (size_t)__MAX_BYTES && new_sz > (size_t)__MAX_BYTES) {
        return(realloc(p, new_sz));
    }
    if (ROUND_UP(old_sz) == ROUND_UP(new_sz))
        return p;

    result = Allocate(new_sz);
    copy_sz = new_sz > old_sz? old_sz : new_sz;
    memcpy(result, p, copy_sz);
    Deallocate(p, old_sz);
    return result;
}
```

- 【STL空间配置的缺点】

1：内碎片问题

2：二级空间配置器的内存什么时候还给操作系统？

- 空间配置器分配空间的初始化和拷贝问题的算法



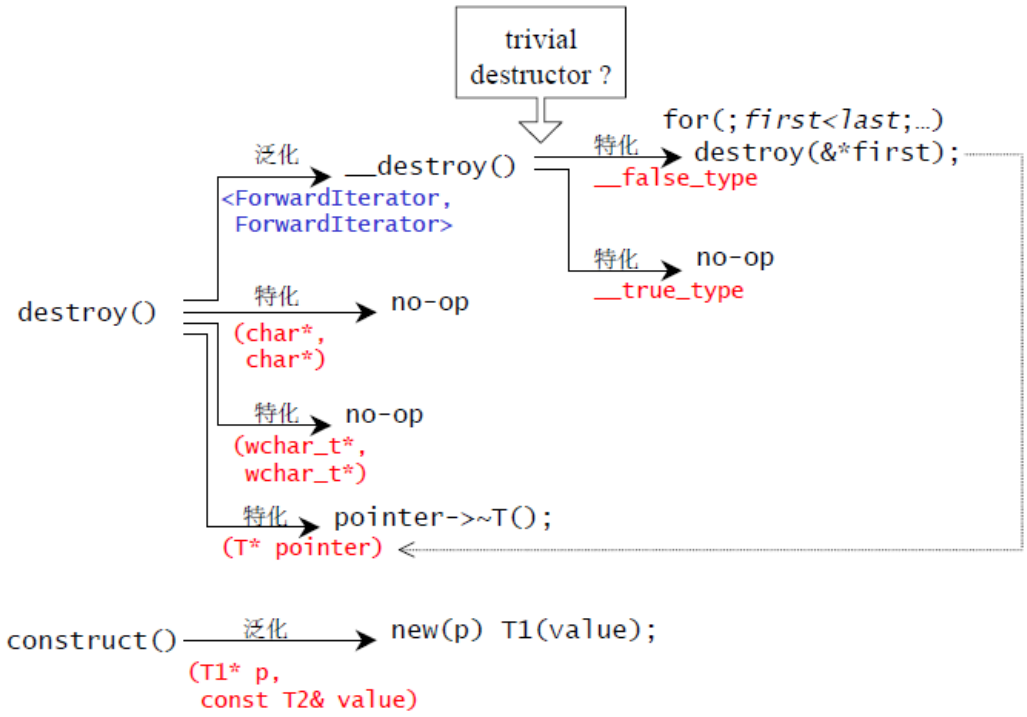




圖 2-1 construct() 和 drestroy() 示意。

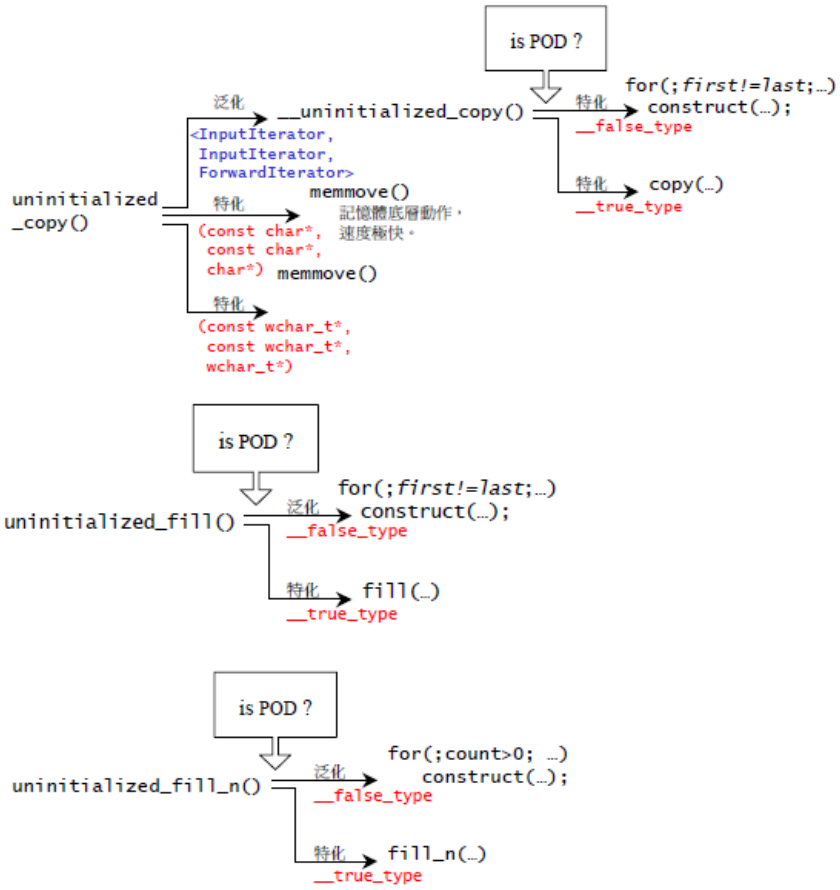



**圖 2-8** 三個記憶體基本函式的泛型版本與特化版本。