# 学习大纲

|  |
| --- |
|  |
|  |
|  |

## 在内存在环境变量是一个接着一个的,并不是一行一行的.

|  |
| --- |
|  |

# API使用什么

## malloc函数的语法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 分配内存块。 语法 C复制  void \*malloc(  size\_t size  ); 参数 *size* 要分配的字节数。 返回值 **malloc** 会返回指向已分配空间的 void 指针，如果可用内存不足，则返回 NULL。 若要返回指向类型而非 **void** 的指针，请在返回值上使用类型转换。 返回值指向的存储空间会适当对齐，以存储对齐要求小于或等于基本对齐要求的任意对象类型。 （在 Visual C++ 中，基本对齐是 **double** 或 8 个字节所需的对齐。在面向 64 位平台的代码中，是 16 个字节。）使用 [\_aligned\_malloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/aligned-malloc?view=msvc-170) 为具有更大对齐要求的对象分配存储空间，例如 SSE 类型 [\_\_m128](https://learn.microsoft.com/zh-cn/cpp/cpp/m128?view=msvc-170) 和 **\_\_m256**，以及使用 \_\_declspec(align( n ))（其中 **n** 大于 8）声明的类型。 如果 *size* 为 0，则 **malloc** 在堆中分配零长度的项并向该项返回有效的指针。 即使请求的内存量较小，也要始终检查 **malloc** 的返回值。 注解 **malloc** 函数分配至少为 *size* 个字节的内存块。 由于对齐和维护信息所需的空间，该块可能大于 *size* 个字节。  如果内存分配失败或请求的内存量超过 \_HEAP\_MAXREQ，**malloc** 会将 errno 设置为 ENOMEM。 有关此错误代码和其他错误代码的信息，请参阅 [errno、\_doserrno、\_sys\_errlist 和 \_sys\_nerr](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/errno-doserrno-sys-errlist-and-sys-nerr?view=msvc-170)。  启动代码使用 **malloc** 为 **\_environ**、*envp* 和 *argv* 变量分配存储空间。 以下函数及其宽字符对应项也调用 **malloc**。  [calloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/calloc?view=msvc-170) [\_exec 函数](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/exec-wexec-functions?view=msvc-170) [fgetc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fgetc-fgetwc?view=msvc-170) [\_fgetchar](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fgetc-fgetwc?view=msvc-170) [fgets](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fgets-fgetws?view=msvc-170) [fprintf](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fprintf-fprintf-l-fwprintf-fwprintf-l?view=msvc-170) [fputc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fputc-fputwc?view=msvc-170) [\_fputchar](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fputc-fputwc?view=msvc-170) [fputs](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fputs-fputws?view=msvc-170) [fread](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fread?view=msvc-170)  [fscanf](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fscanf-fscanf-l-fwscanf-fwscanf-l?view=msvc-170) [fseek](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fseek-fseeki64?view=msvc-170) [fsetpos](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fsetpos?view=msvc-170) [\_fullpath](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fullpath-wfullpath?view=msvc-170) [fwrite](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/fwrite?view=msvc-170) [getc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/getc-getwc?view=msvc-170) [getchar](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/getc-getwc?view=msvc-170) [\_getcwd](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/getcwd-wgetcwd?view=msvc-170) [\_getdcwd](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/getcwd-wgetcwd?view=msvc-170) [gets](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/gets-getws?view=msvc-170)  [\_getw](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/getw?view=msvc-170) [\_popen](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/popen-wpopen?view=msvc-170) [printf](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/printf-printf-l-wprintf-wprintf-l?view=msvc-170) [putc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/putc-putwc?view=msvc-170) [putchar](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/putc-putwc?view=msvc-170) [\_putenv](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/putenv-wputenv?view=msvc-170) [puts](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/puts-putws?view=msvc-170) [\_putw](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/putw?view=msvc-170) [scanf](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/scanf-scanf-l-wscanf-wscanf-l?view=msvc-170)  [\_searchenv](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/searchenv-wsearchenv?view=msvc-170) [setvbuf](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setvbuf?view=msvc-170) [\_spawn 函数](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/spawn-wspawn-functions?view=msvc-170) [\_strdup](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/strdup-wcsdup-mbsdup?view=msvc-170) [system](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/system-wsystem?view=msvc-170) [\_tempnam](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/tempnam-wtempnam-tmpnam-wtmpnam?view=msvc-170) [ungetc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/ungetc-ungetwc?view=msvc-170) [vfprintf](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/vfprintf-vfprintf-l-vfwprintf-vfwprintf-l?view=msvc-170) [vprintf](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/vprintf-vprintf-l-vwprintf-vwprintf-l?view=msvc-170)  C++ [\_set\_new\_mode](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/set-new-mode?view=msvc-170) 函数将为 **malloc** 设置新的处理程序模式。 新的处理程序模式将指示 **malloc** 是否在失败时调用由 [\_set\_new\_handler](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/set-new-handler?view=msvc-170) 设置的新处理程序例程。 默认情况下，**malloc** 在失败时不调用新的处理程序例程来分配内存。 可以替代此默认行为，以便在 **malloc** 无法分配内存时，**malloc** 将以 **new** 运算符由于相同原因失败时的同一方法调用新的处理程序例程。 要替代默认值，请提前在程序中调用 \_set\_new\_mode(1) 或与 NEWMODE.OBJ 链接（请参阅[链接选项](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/link-options?view=msvc-170)）。  当应用程序与调试版的 C 运行时库链接时，**malloc** 将解析为 [\_malloc\_dbg](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/malloc-dbg?view=msvc-170)。 有关堆在调试过程中如何托管的详细信息，请参阅 [CRT 调试堆详细信息](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/crt-debug-heap-details?view=msvc-170)。  **malloc** 标记了 \_\_declspec(noalias) 和 \_\_declspec(restrict)。 这些属性意味着该函数保证不会修改全局变量，并且返回的指针不使用别名。 有关详细信息，请参阅 [noalias](https://learn.microsoft.com/zh-cn/cpp/cpp/noalias?view=msvc-170) 和 [restrict](https://learn.microsoft.com/zh-cn/cpp/cpp/restrict?view=msvc-170)。  默认情况下，此函数的全局状态范围限定为应用程序。 若要更改此行为，请参阅 [CRT 中的全局状态](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/global-state?view=msvc-170)。 要求 展开表   | **例程** | **必需的标头** | | --- | --- | | **malloc** | <stdlib.h> 和 <malloc.h> |   有关兼容性的详细信息，请参阅 [兼容性](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/compatibility?view=msvc-170)。 库 [C 运行时库](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/crt-library-features?view=msvc-170)的所有版本。 示例 C复制  // crt\_malloc.c  // This program allocates memory with  // malloc, then frees the memory with free.  #include <stdlib.h> // For \_MAX\_PATH definition  #include <stdio.h>  #include <malloc.h>  int main( void )  {  char \*string;  // Allocate space for a path name  string = malloc( \_MAX\_PATH );  // In a C++ file, explicitly cast malloc's return. For example,  // string = (char \*)malloc( \_MAX\_PATH );  if( string == NULL )  printf( "Insufficient memory available\n" );  else  {  printf( "Memory space allocated for path name\n" );  free( string );  printf( "Memory freed\n" );  }  }  Output复制  Memory space allocated for path name  Memory freed 另请参阅 [内存分配](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/memory-allocation?view=msvc-170) [calloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/calloc?view=msvc-170) [free](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/free?view=msvc-170) [realloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/realloc?view=msvc-170) [\_aligned\_malloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/aligned-malloc?view=msvc-170) |

## calloc函数的语法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 使用初始化为 0 的元素分配内存中的数组。 语法 C复制  void \*calloc(  size\_t number,  size\_t size  ); 参数 *number* 元素数量。  *size* 每个元素的长度（以字节为单位）。 返回值 **calloc** 返回指向已分配空间的指针。 返回值指向的存储空间与任何类型的对象的存储适当对齐。 若要获取指向类型而非 **void** 的指针，请在返回值中使用类型转换。 备注 **calloc** 函数为 *number* 元素的数组分配存储空间，每个长度为 *size* 字节。 将每个元素初始化为 0。  如果内存分配失败或请求的内存量超过 \_HEAP\_MAXREQ，**calloc** 会将 errno 设置为 ENOMEM。 有关此错误代码和其他错误代码的信息，请参阅 [errno、\_doserrno、\_sys\_errlist 和 \_sys\_nerr](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/errno-doserrno-sys-errlist-and-sys-nerr?view=msvc-170)。  在 Microsoft 实现中，如果 *number* 或 *size* 为零，则 **calloc** 返回指向非零大小的已分配块的指针。 尝试通过返回的指针读取或写入会导致未定义的行为。  **calloc** 使用 C++ [\_set\_new\_mode](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/set-new-mode?view=msvc-170) 函数设置新的处理程序模式。 新的处理程序模式将指示 **calloc** 是否在失败时调用由 [\_set\_new\_handler](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/set-new-handler?view=msvc-170) 设置的新处理程序例程。 默认情况下，**calloc** 在失败时不调用新的处理程序例程来分配内存。 可以替代此默认行为，以便在 **calloc** 无法分配内存时，它可以调用新的处理程序例程，方法与 **new** 运算符出于相同原因无法分配内存时所执行的操作一样。 若要替代默认值，请在程序的早期调用：  C复制  \_set\_new\_mode(1);  或链接到 *NEWMODE.OBJ*（请参阅[链接选项](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/link-options?view=msvc-170)）。  当应用程序与调试版的 C 运行时库链接时，**calloc** 将解析为 [\_calloc\_dbg](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/calloc-dbg?view=msvc-170)。 有关在调试过程中如何托管堆的详细信息，请参阅 [CRT 调试堆](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/crt-debug-heap-details?view=msvc-170)。  **calloc** 被标记为 \_\_declspec(noalias) 和 \_\_declspec(restrict)，也就是说确保该函数不能修改全局变量，并且指针返回不使用别名。 有关详细信息，请参阅 [noalias](https://learn.microsoft.com/zh-cn/cpp/cpp/noalias?view=msvc-170) 和 [restrict](https://learn.microsoft.com/zh-cn/cpp/cpp/restrict?view=msvc-170)。  默认情况下，此函数的全局状态范围限定为应用程序。 若要更改此行为，请参阅 [CRT 中的全局状态](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/global-state?view=msvc-170)。 要求 展开表   | **例程** | **必需的标头** | | --- | --- | | **calloc** | <stdlib.h> 和 <malloc.h> |   有关兼容性的详细信息，请参阅 [兼容性](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/compatibility?view=msvc-170)。 示例 C  // crt\_calloc.c  // This program uses calloc to allocate space for  // 40 long integers. It initializes each element to zero.  #include <stdio.h>  #include <malloc.h>  int main( void )  {  long \*buffer;  buffer = (long \*)calloc( 40, sizeof( long ) );  if( buffer != NULL )  printf( "Allocated 40 long integers\n" );  else  printf( "Can't allocate memory\n" );  free( buffer );  }  Output复制  Allocated 40 long integers 另请参阅 [内存分配](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/memory-allocation?view=msvc-170) [free](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/free?view=msvc-170) [malloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/malloc?view=msvc-170) [realloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/realloc?view=msvc-170) |

## realloc函数的用法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 重新分配内存块。 语法 C  void \*realloc(  void \*memblock,  size\_t size  ); 参数 *memblock* 指向之前已分配内存块的指针。  *size* 新大小（字节）。 返回值 **realloc** 将返回指向重新分配的（并且可能已移动的）内存块的 **void** 指针。  如果没有足够可用的内存将块扩展到给定大小，则原始块将保持不变，并返回 NULL。  如果 *size* 为零，则释放由 *memblock* 指向的块；返回值为 NULL，而 *memblock* 仍指向已释放的块。  返回值将指向适当对齐任何类型的对象的存储的存储空间。 若要获取指向类型而非 **void** 的指针，请在返回值中使用类型转换。 备注 **备注**  尚未为实现 C17 行为而更新 **realloc**，因为新行为与 Windows 操作系统不兼容。  **realloc** 函数更改已分配内存块的大小。 *memblock* 参数指向内存块的开头。 如果 *memblock* 为 NULL，则 **realloc** 与 **malloc** 的行为相同，并分配一个 *size* 字节的新块。 如果 *memblock* 不为 NULL，则它应是指向以前调用 **calloc**、**malloc** 或 **realloc** 所返回的指针。  *size* 参数提供块的新大小（字节）。 块的内容不随其新旧大小而更改，尽管新块可以在不同的位置。 因为新块可以在新的内存位置，所以由 **realloc** 返回的指针并非一定是指向通过 *memblock* 参数传递的指针。 如果存在缓冲区增长，则 **realloc** 不会将新分配的内存清零。  如果内存分配失败或请求的内存量超过 \_HEAP\_MAXREQ，则 **realloc** 将 errno 设置为 ENOMEM。 有关此错误代码和其他错误代码的信息，请参阅 [errno、\_doserrno、\_sys\_errlist 和 \_sys\_nerr](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/errno-doserrno-sys-errlist-and-sys-nerr?view=msvc-170)。  **realloc** 调用 **malloc** 以使用 C++ [\_set\_new\_mode](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/set-new-mode?view=msvc-170) 函数设置新的处理程序模式。 新的处理程序模式将指示 **malloc** 是否在失败时调用由 [\_set\_new\_handler](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/set-new-handler?view=msvc-170) 设置的新处理程序例程。 默认情况下，**malloc** 在失败时不调用新的处理程序例程来分配内存。 可以替代此默认行为，以便在 **realloc** 无法分配内存时，**malloc** 将以 **new** 运算符由于相同原因失败时的同一方法调用新的处理程序例程。 若要替代默认值，请在程序的早期调用：  C复制  \_set\_new\_mode(1);  或链接到 NEWMODE.OBJ（请参阅[链接选项](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/link-options?view=msvc-170)）。  当应用程序与调试版的 C 运行时库链接时，**realloc** 将解析为 [\_realloc\_dbg](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/realloc-dbg?view=msvc-170)。 有关在调试过程中如何托管堆的详细信息，请参阅 [CRT 调试堆](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/crt-debug-heap-details?view=msvc-170)。  **realloc** 被标记为 \_\_declspec(noalias) 和 \_\_declspec(restrict)，也就是说确保该函数不能修改全局变量，并且指针返回不使用别名。 有关详细信息，请参阅 [noalias](https://learn.microsoft.com/zh-cn/cpp/cpp/noalias?view=msvc-170) 和 [restrict](https://learn.microsoft.com/zh-cn/cpp/cpp/restrict?view=msvc-170)。  默认情况下，此函数的全局状态范围限定为应用程序。 若要更改此行为，请参阅 [CRT 中的全局状态](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/global-state?view=msvc-170)。 要求 展开表   | **例程** | **必需的标头** | | --- | --- | | **realloc** | <stdlib.h> 和 <malloc.h> |   有关兼容性的详细信息，请参阅 [兼容性](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/compatibility?view=msvc-170)。 示例 C复制  // crt\_realloc.c  // This program allocates a block of memory for  // buffer and then uses \_msize to display the size of that  // block. Next, it uses realloc to expand the amount of  // memory used by buffer and then calls \_msize again to  // display the new amount of memory allocated to buffer.  #include <stdio.h>  #include <malloc.h>  #include <stdlib.h>  int main( void )  {  long \*buffer, \*oldbuffer;  size\_t size;  if( (buffer = (long \*)malloc( 1000 \* sizeof( long ) )) == NULL )  exit( 1 );  size = \_msize( buffer );  printf\_s( "Size of block after malloc of 1000 longs: %u\n", size );  // Reallocate and show new size:  oldbuffer = buffer; // save pointer in case realloc fails  if( (buffer = realloc( buffer, size + (1000 \* sizeof( long )) ))  == NULL )  {  free( oldbuffer ); // free original block  exit( 1 );  }  size = \_msize( buffer );  printf\_s( "Size of block after realloc of 1000 more longs: %u\n",  size );  free( buffer );  exit( 0 );  }  Output复制  Size of block after malloc of 1000 longs: 4000  Size of block after realloc of 1000 more longs: 8000 另请参阅 [内存分配](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/memory-allocation?view=msvc-170) [calloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/calloc?view=msvc-170) [free](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/free?view=msvc-170) [malloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/malloc?view=msvc-170) |

## free函数的语法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 解除分配或释放内存块。 语法 C复制  void free(  void \*memblock  ); 参数 *memblock* 要释放的以前分配的内存块。 注解 **free** 函数释放以前通过调用 **calloc**、**malloc** 或 **realloc** 分配的内存块 (*memblock*)。 已释放的字节数等于分配块时（对于 **realloc** 则为重新分配）请求的字节数。 如果 *memblock* 是 NULL，会忽略指针并立即返回 **free**。 尝试释放无效指针（指向并非由 **calloc**、**malloc** 或 **realloc** 分配的内存块的指针）可能会影响后续分配请求，并导致错误。  如果在释放内存时发生错误，则根据操作系统中关于错误性质的信息设置 errno。 有关详细信息，请参阅[errno, \_doserrno, \_sys\_errlist和\_sys\_nerr](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/errno-doserrno-sys-errlist-and-sys-nerr?view=msvc-170).  释放内存块后，[\_heapmin](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/heapmin?view=msvc-170) 通过合并未使用的区域并将其释放回到操作系统，将堆上的可用内存量降至最低。 未释放给操作系统的已释放内存还原到可用池，并且可用于重新分配。  当应用程序与调试版的 C 运行时库链接时，**free** 将解析为 [\_free\_dbg](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/free-dbg?view=msvc-170)。 若要详细了解如何在调试过程中托管堆，请参阅 [CRT 调试堆](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/crt-debug-heap-details?view=msvc-170)。  **free** 标记为 \_\_declspec(noalias)，这表示该函数保证不会修改全局变量。 有关详细信息，请参阅 [noalias](https://learn.microsoft.com/zh-cn/cpp/cpp/noalias?view=msvc-170)。  若要释放使用 [\_malloca](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/malloca?view=msvc-170) 分配的内存，请使用 [\_freea](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/freea?view=msvc-170)。  默认情况下，此函数的全局状态范围限定为应用程序。 若要更改此行为，请参阅 [CRT 中的全局状态](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/global-state?view=msvc-170)。 要求 展开表   | **函数** | **必需的标头** | | --- | --- | | **free** | <stdlib.h> 和 <malloc.h> |   有关兼容性的详细信息，请参阅 [兼容性](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/compatibility?view=msvc-170)。 示例 请参阅 [malloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/malloc?view=msvc-170) 的示例。 另请参阅 [内存分配](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/memory-allocation?view=msvc-170) [\_alloca](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/alloca?view=msvc-170) [calloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/calloc?view=msvc-170) [malloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/malloc?view=msvc-170) [realloc](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/realloc?view=msvc-170) [\_free\_dbg](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/free-dbg?view=msvc-170) [\_heapmin](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/heapmin?view=msvc-170) [\_freea](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/freea?view=msvc-170) |

## GetEnvironmentStrings函数的语法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 检索当前进程的环境变量。 语法 C++复制  LPCH GetEnvironmentStrings(); 返回值 如果函数成功，则返回值是指向当前进程的环境块的指针。  如果函数失败，则返回值为 NULL。 注解 **GetEnvironmentStrings** 函数返回指向内存块的指针，该内存块包含调用进程的环境变量 (系统和用户环境变量) 。  每个环境块包含以下格式的环境变量：  Var1=Value1\0 Var2=Value2\0 Var3=Value3\0 ... VarN=ValueN\0\0  环境变量的名称不能包含等号 (=) 。  将此内存视为只读内存;不要直接修改它。 若要添加或更改环境变量，请使用 [GetEnvironmentVariable](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-getenvironmentvariable) 和 [SetEnvironmentVariable](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-setenvironmentvariable) 函数。  当不再需要 **GetEnvironmentStrings** 返回的块时，应通过调用 [FreeEnvironmentStrings](https://learn.microsoft.com/zh-cn/windows/desktop/api/processenv/nf-processenv-freeenvironmentstringsa) 函数来释放该块。  请注意，此函数的 ANSI 版本 **GetEnvironmentStringsA** 返回 OEM 字符。 示例 有关示例，请参阅 [更改环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/changing-environment-variables)。 要求  |  |  | | --- | --- | | **标头** | processenv.h (包括 Windows Server 2003 上的 Windows.h、Windows Vista、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/environment-variables)  [FreeEnvironmentStrings](https://learn.microsoft.com/zh-cn/windows/desktop/api/processenv/nf-processenv-freeenvironmentstringsa)  [GetEnvironmentVariable](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-getenvironmentvariable)  [SetEnvironmentVariable](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-setenvironmentvariable) |

## GetEnvironmentVariable函数的语法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 从调用进程的环境块检索指定变量的内容。 语法 C++  DWORD GetEnvironmentVariable(  [in, optional] LPCTSTR lpName,  [out, optional] LPTSTR lpBuffer,  [in] DWORD nSize  ); 参数 [in, optional] lpName  环境变量的名称。  [out, optional] lpBuffer  指向缓冲区的指针，该缓冲区以 null 结尾的字符串的形式接收指定环境变量的内容。 环境变量的最大大小限制为 32,767 个字符，包括 null 终止字符。  [in] nSize  *lpBuffer* 参数指向的缓冲区的大小，包括 null 终止符，以字符为单位。 返回值 如果函数成功，则返回值是 *lpBuffer* 指向的缓冲区中存储的字符数，不包括终止 null 字符。  如果 *lpBuffer* 不够大来保存数据，则返回值是保存字符串及其终止 null 字符所需的缓冲区大小（以字符为单位）， *lpBuffer* 的内容未定义。  如果函数失败，则返回值为零。 如果在环境块中找不到指定的环境变量， [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 将返回ERROR\_ENVVAR\_NOT\_FOUND。 注解 此函数可以检索系统环境变量或用户环境变量。 示例 有关示例，请参阅 [更改环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/changing-environment-variables)。 要求  |  |  | | --- | --- | | **标头** | winbase.h (包括 Windows Server 2003、Windows Vista、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/environment-variables)  [GetEnvironmentStrings](https://learn.microsoft.com/zh-cn/windows/win32/api/processenv/nf-processenv-getenvironmentstrings)  [SetEnvironmentVariable](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-setenvironmentvariable) |

## FreeEnvironmentStrings函数的用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 释放环境字符串块。 语法 C++  BOOL FreeEnvironmentStringsA(  LPCH penv  ); 参数 penv  指向环境字符串块的指针。 必须通过调用 [GetEnvironmentStrings](https://learn.microsoft.com/zh-cn/windows/desktop/api/rrascfg/nf-rrascfg-ieapproviderconfig-initialize) 函数来获取指向 块的指针。 返回值 如果该函数成功，则返回值为非零值。  如果函数失败，则返回值为零 若要获取扩展错误信息，请调用 [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror)。 注解 如果使用 [了 AnSI 版本的 GetEnvironmentStrings](https://learn.microsoft.com/zh-cn/windows/desktop/api/rrascfg/nf-rrascfg-ieapproviderconfig-initialize)，请务必使用 **AnSI 版本的 FreeEnvironmentStrings**。 同样，如果使用 **了 GetEnvironmentStrings** 的 Unicode 版本，请务必使用 **Unicode 版本的 FreeEnvironmentStrings**。 示例 有关示例，请参阅 [更改环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/changing-environment-variables)。  **备注**  processenv.h 标头将 FreeEnvironmentStrings 定义为别名，该别名根据 UNICODE 预处理器常量的定义自动选择此函数的 ANSI 或 Unicode 版本。 将非特定编码别名与非非特定编码的代码混合使用可能会导致不匹配，从而导致编译或运行时错误。 有关详细信息，请参阅 [**函数原型的约定**](https://learn.microsoft.com/zh-cn/windows/win32/intl/conventions-for-function-prototypes)。 要求  |  |  | | --- | --- | | **标头** | processenv.h (包括 Windows Server 2003、Windows Vista、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/environment-variables)  [GetEnvironmentStrings](https://learn.microsoft.com/zh-cn/windows/desktop/api/rrascfg/nf-rrascfg-ieapproviderconfig-initialize) |

## GetProcessHeap函数用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 检索调用进程的**默认堆的句柄**。 然后，可以在对堆函数的后续调用中使用此句柄。 语法 C++  HANDLE GetProcessHeap(); 返回值 如果函数成功，则返回值是调用进程的堆的句柄。  如果函数失败，则返回值为 NULL。 要获得更多的错误信息，请调用 GetLastError。 注解 **GetProcessHeap** 函数获取调用进程的默认堆的句柄。 进程可以使用此句柄从进程堆分配内存，而无需先使用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heapcreate) 函数创建专用堆。  **Windows Server 2003 和 Windows XP：**若要为进程的默认堆启用低碎片堆，请使用 **GetProcessHeap** 返回的句柄调用 [HeapSetInformation](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapsetinformation) 函数。 示例 有关示例，请参阅 [获取进程堆](https://learn.microsoft.com/zh-cn/windows/desktop/Memory/getting-process-heaps)。 要求  |  |  | | --- | --- | | **标头** | heapapi.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [堆函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/heap-functions)  [HeapCreate](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heapcreate)  [内存管理函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/memory-management-functions)  [VBS enclave 中可用的 Vertdll API](https://learn.microsoft.com/zh-cn/windows/win32/trusted-execution/enclaves-available-in-vertdll) |

## HeapAlloc函数的语法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 从堆中分配内存块。 分配的内存不可移动。 语法 C++  DECLSPEC\_ALLOCATOR LPVOID HeapAlloc(  [in] HANDLE hHeap,  [in] DWORD dwFlags,  [in] SIZE\_T dwBytes  ); 参数 [in] hHeap  要从中分配内存的堆的句柄。 此句柄由 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 或 [GetProcessHeap](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-getprocessheap) 函数返回。  [in] dwFlags  堆分配选项。 指定这些值中的任何一个都将替代使用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 创建堆时指定的相应值。 此参数可使用以下一个或多个值。  展开表   |  |  | | --- | --- | | **值** | **含义** | | **HEAP\_GENERATE\_EXCEPTIONS**  0x00000004 | 系统将引发异常以指示函数失败（例如内存不足情况），而不是返回 **NULL**。  若要确保为此函数的所有调用生成异常，请在调用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 时指定**HEAP\_GENERATE\_EXCEPTIONS**。 在这种情况下，无需在此函数调用中额外指定 **HEAP\_GENERATE\_EXCEPTIONS** 。 | | **HEAP\_NO\_SERIALIZE**  0x00000001 | 序列化访问将不用于此分配。  有关详细信息，请参阅“备注”。  若要确保禁用对此函数的所有调用的序列化访问，请在调用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 中指定**HEAP\_NO\_SERIALIZE**。 在这种情况下，无需在此函数调用中额外指定 **HEAP\_NO\_SERIALIZE** 。  访问进程的默认堆时，不应指定此值。 系统可能会在应用程序的进程中创建其他线程，例如同时访问进程的默认堆的 CTRL+C 处理程序。 | | **HEAP\_ZERO\_MEMORY**  0x00000008 | 分配的内存将初始化为零。 否则，内存不会初始化为零。 |   [in] dwBytes  要分配的字节数。  如果 *hHeap* 参数指定的堆是“不可增长的”堆， *则 dwBytes* 必须小于 0x7FFF8。 可以通过使用非零值调用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 函数来创建不可增长的堆。 返回值 如果函数成功，则返回值是指向已分配内存块的指针。  如果函数失败并且您尚未指定 **HEAP\_GENERATE\_EXCEPTIONS**，则返回值为 **NULL**。  如果函数失败并且已指定 **HEAP\_GENERATE\_EXCEPTIONS**，则函数可能会生成下表中列出的任一异常。 特定例外取决于堆损坏的性质。 有关详细信息，请参阅 [GetExceptionCode](https://learn.microsoft.com/zh-cn/windows/desktop/Debug/getexceptioncode)。  展开表   |  |  | | --- | --- | | **异常代码** | **说明** | | **STATUS\_NO\_MEMORY** | 由于缺少可用内存或堆损坏，分配尝试失败。 | | **STATUS\_ACCESS\_VIOLATION** | 由于堆损坏或函数参数不正确，分配尝试失败。 |   如果函数失败，则它不会调用 [SetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-setlasterror)。 应用程序无法调用 [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 以获取扩展错误信息。 注解 如果 **HeapAlloc** 函数成功，它将分配至少请求的内存量。  若要从进程的默认堆分配内存，请将 **HeapAlloc** 与 [GetProcessHeap](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-getprocessheap) 函数返回的句柄一起使用。  若要释放 **HeapAlloc** 分配的内存块，请使用 [HeapFree](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapfree) 函数。  **HeapAlloc** 分配的内存不可移动。 **HeapAlloc** 返回的地址在释放或重新分配内存块之前有效;内存块不需要锁定。 由于系统无法压缩专用堆，因此它可能会碎片化。  **HeapAlloc** 返回的内存对齐方式在 WinNT.h 中**MEMORY\_ALLOCATION\_ALIGNMENT**：  C++复制  #if defined(\_WIN64) || defined(\_M\_ALPHA)  #define MEMORY\_ALLOCATION\_ALIGNMENT 16  #else  #define MEMORY\_ALLOCATION\_ALIGNMENT 8  #endif  以各种分配大小分配大量内存的应用程序可以使用 [低碎片堆](https://learn.microsoft.com/zh-cn/windows/desktop/Memory/low-fragmentation-heap) 来减少堆碎片。  当两个或多个线程尝试从同一堆同时分配或释放块时，序列化可确保相互排斥。 序列化的性能成本很小，但每当多个线程从同一个堆分配和释放内存时，必须使用它。 设置 **HEAP\_NO\_SERIALIZE** 值可消除堆上的相互排斥。 如果不进行序列化，使用同一堆句柄的两个或多个线程可能会尝试同时分配或释放内存，这可能会导致堆损坏。 因此，只能在以下情况下安全地使用 **HEAP\_NO\_SERIALIZE** 值：   * 进程只有一个线程。 * 进程有多个线程，但只有一个线程调用特定堆的堆函数。 * 进程具有多个线程，应用程序提供自己的机制，用于对特定堆进行相互排斥。  示例 有关示例，请参阅 [AWE 示例](https://learn.microsoft.com/zh-cn/windows/win32/Memory/awe-example)。 要求  |  |  | | --- | --- | | **标头** | heapapi.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [堆函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/heap-functions)  [HeapFree](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heapfree)  [HeapReAlloc](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heaprealloc)  [内存管理函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/memory-management-functions)  [VBS enclave 中可用的 Vertdll API](https://learn.microsoft.com/zh-cn/windows/win32/trusted-execution/enclaves-available-in-vertdll) |

## HeapReAlloc 函数语法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 从堆中重新分配内存块。 使用此函数可以调整内存块的大小并更改其他内存块属性。 分配的内存不可移动。 语法 C++复制  DECLSPEC\_ALLOCATOR LPVOID HeapReAlloc(  [in] HANDLE hHeap,  [in] DWORD dwFlags,  [in] \_Frees\_ptr\_opt\_ LPVOID lpMem,  [in] SIZE\_T dwBytes  ); 参数 [in] hHeap  要从中重新分配内存的堆的句柄。 此句柄是由 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 或 [GetProcessHeap](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-getprocessheap) 函数返回的 。  [in] dwFlags  堆重新分配选项。 使用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 函数创建堆时，指定值将替代 *flOptions* 参数中指定的相应值。 此参数可使用以下一个或多个值。  展开表   |  |  | | --- | --- | | **值** | **含义** | | **HEAP\_GENERATE\_EXCEPTIONS**  0x00000004 | 操作系统引发异常以指示函数失败（例如内存不足的情况），而不是返回 **NULL**。  若要确保为此函数的所有调用生成异常，请在对 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 的调用中指定**HEAP\_GENERATE\_EXCEPTIONS**。 在这种情况下，无需在此函数调用中额外指定 **HEAP\_GENERATE\_EXCEPTIONS** 。 | | **HEAP\_NO\_SERIALIZE**  0x00000001 | 不会使用序列化访问。 有关详细信息，请参阅“备注”。  若要确保对此函数的所有调用禁用序列化访问，请在对 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 的调用中指定**HEAP\_NO\_SERIALIZE**。 在这种情况下，无需在此函数调用中额外指定 **HEAP\_NO\_SERIALIZE** 。  访问进程堆时不应指定此值。 系统可能会在应用程序的进程中创建其他线程，例如同时访问进程堆的 CTRL+C 处理程序。 | | **HEAP\_REALLOC\_IN\_PLACE\_ONLY**  0x00000010 | 重新分配内存块时不能移动。 如果未指定此值，该函数可能会将块移动到新位置。 如果指定了此值，并且无法在不移动的情况下调整块的大小，则函数将失败，使原始内存块保持不变。 | | **HEAP\_ZERO\_MEMORY**  0x00000008 | 如果重新分配请求的大小更大，则超出原始大小的其他内存区域将初始化为零。 内存块的内容与其原始大小一起不受影响。 |   [in] lpMem  指向函数重新分配的内存块的指针。 此指针由先前对 [HeapAlloc](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapalloc) 或 **HeapReAlloc** 函数的调用返回。  [in] dwBytes  内存块的新大小（以字节为单位）。 使用此函数可以增加或减小内存块的大小。  如果 *hHeap* 参数指定的堆是“不可增长”堆， *则 dwBytes* 必须小于 0x7FFF8。 可以通过调用具有非零值的 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 函数来创建不可增长的堆。 返回值 如果函数成功，则返回值是指向重新分配的内存块的指针。  如果函数失败，并且您尚未指定 **HEAP\_GENERATE\_EXCEPTIONS**，则返回值为 **NULL**。  如果函数失败，并且已指定 **HEAP\_GENERATE\_EXCEPTIONS**，该函数可能会生成下表中列出的任一异常。 有关详细信息，请参阅 [GetExceptionCode](https://learn.microsoft.com/zh-cn/windows/desktop/Debug/getexceptioncode)。  展开表   |  |  | | --- | --- | | **异常代码** | **说明** | | **STATUS\_NO\_MEMORY** | 由于缺少可用内存或堆损坏，分配尝试失败。 | | **STATUS\_ACCESS\_VIOLATION** | 由于堆损坏或函数参数不正确，分配尝试失败。 |   **HeapReAlloc** 返回的内存对齐方式在 WinNT.h **中MEMORY\_ALLOCATION\_ALIGNMENT**：  C++  #if defined(\_WIN64) || defined(\_M\_ALPHA)  #define MEMORY\_ALLOCATION\_ALIGNMENT 16  #else  #define MEMORY\_ALLOCATION\_ALIGNMENT 8  #endif  如果函数失败，则不调用 [SetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-setlasterror)。 应用程序无法调用 [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 以获取扩展错误信息。 注解 如果 **HeapReAlloc** 成功，它将至少分配请求的内存量。  如果 **HeapReAlloc** 失败，则不会释放原始内存，并且原始句柄和指针仍然有效。  可以保证 **HeapReAlloc** 保留要重新分配的内存的内容，即使新内存是在不同的位置分配的。 保留内存内容的过程涉及可能非常耗时的内存复制操作。  若要释放 **HeapReAlloc** 分配的内存块，请使用 [HeapFree](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heapfree) 函数。  当两个或多个线程尝试从同一堆同时分配或释放块时，序列化可确保相互排斥。 序列化的性能成本很小，但每当多个线程从同一堆分配和释放内存时，必须使用它。 设置 **HEAP\_NO\_SERIALIZE** 值可消除堆上的相互排斥。 如果不进行序列化，使用同一堆句柄的两个或多个线程可能会尝试同时分配或释放内存，这可能会导致堆损坏。 因此，只能在以下情况下安全地使用 **HEAP\_NO\_SERIALIZE** 值：   * 进程只有一个线程。 * 进程有多个线程，但只有一个线程调用特定堆的堆函数。 * 进程具有多个线程，应用程序为特定堆提供自身的相互排斥机制。  要求  |  |  | | --- | --- | | **标头** | heapapi.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [堆函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/heap-functions)  [HeapAlloc](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heapalloc)  [HeapFree](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heapfree)  [内存管理函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/memory-management-functions)  [VBS enclave 中可用的 Vertdll API](https://learn.microsoft.com/zh-cn/windows/win32/trusted-execution/enclaves-available-in-vertdll) |

## HeapFree函数的语法

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 释放由 [HeapAlloc 或 HeapReAlloc](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapalloc) 函数从堆分配的内存块。 语法 C++复制  BOOL HeapFree(  [in] HANDLE hHeap,  [in] DWORD dwFlags,  [in] \_Frees\_ptr\_opt\_ LPVOID lpMem  ); 参数 [in] hHeap  要释放其内存块的堆的句柄。 此句柄由 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 或 [GetProcessHeap](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-getprocessheap) 函数返回。  [in] dwFlags  堆免费选项。 使用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 函数创建堆时，指定以下值将替代 *flOptions* 参数中指定的相应值。  展开表   |  |  | | --- | --- | | **值** | **含义** | | **HEAP\_NO\_SERIALIZE**  0x00000001 | 不会使用序列化访问。 有关详细信息，请参阅“备注”。  若要确保对此函数的所有调用禁用序列化访问，请在对 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 的调用中指定**HEAP\_NO\_SERIALIZE**。 在这种情况下，无需在此函数调用中额外指定 **HEAP\_NO\_SERIALIZE** 。  访问进程堆时不要指定此值。 系统可能会在应用程序的进程中创建其他线程，例如同时访问进程堆的 CTRL+C 处理程序。 |   [in] lpMem  指向要释放的内存的指针。 此指针由 [HeapAlloc](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapalloc) 或 [HeapReAlloc](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heaprealloc) 函数返回。 此指针可以为 **NULL**。 返回值 如果该函数成功，则返回值为非零值。  如果函数失败，则返回值为零。 应用程序可以调用 [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 以获取扩展错误信息。 注解 不应以任何方式引用已由 **HeapFree** 释放的内存。 释放该内存后，可能已存在的任何信息将永远消失。 如果需要信息，请不要释放包含该信息的内存。 返回有关内存 (（例如 [HeapSize](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapsize)) ）信息的函数调用可能不会与释放的内存一起使用，因为它们可能会返回虚假数据。 使用同一指针调用 **HeapFree** 两次可能会导致堆损坏，导致对 [HeapAlloc](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapalloc) 的后续调用返回同一指针两次。  当两个或多个线程尝试从同一堆同时分配或释放块时，序列化可确保相互排斥。 序列化的性能成本很小，但每当多个线程从同一堆分配和释放内存时，必须使用它。 设置 **HEAP\_NO\_SERIALIZE** 值可消除堆上的相互排斥。 如果不进行序列化，使用同一堆句柄的两个或多个线程可能会尝试同时分配或释放内存，这可能会导致堆损坏。 因此，只能在以下情况下安全地使用 **HEAP\_NO\_SERIALIZE** 值：   * 进程只有一个线程。 * 进程有多个线程，但只有一个线程调用特定堆的堆函数。 * 进程具有多个线程，应用程序为特定堆提供自身的相互排斥机制。  示例 有关示例，请参阅 [获取进程堆](https://learn.microsoft.com/zh-cn/windows/desktop/Memory/getting-process-heaps)。 要求  |  |  | | --- | --- | | **标头** | heapapi.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [堆函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/heap-functions)  [HeapAlloc](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heapalloc)  [HeapReAlloc](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heaprealloc)  [内存管理函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/memory-management-functions)  [VBS enclave 中可用的 Vertdll API](https://learn.microsoft.com/zh-cn/windows/win32/trusted-execution/enclaves-available-in-vertdll) |

# 演练

## 1.新建一个常规空项目,取名Lesson83-environ,然后添加一个cpp源文件,取名environ.cpp

|  |
| --- |
|  |

## 2.给这个项目添加一个对话框资源

|  |
| --- |
|  |

## 3.给对话框添加如下布局

|  |
| --- |
|  |
|  |

## 4.给我们的程序添加一些创建对话框右键一些基本的命令响应代码如关闭按钮点击等等

|  |
| --- |
| #include<Windows.h>  #include"resource.h"  INT\_PTR CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);  int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE prev,LPTSTR lpCmd,int nShow)  {  DialogBox(hInstance,MAKEINTRESOURCE(IDD\_DIALOG1),NULL,DlgProc);  return 0;  }  INT\_PTR CALLBACK DlgProc(HWND hdlg,UINT msg,WPARAM wParam,LPARAM lParam)  {  BOOL ret = TRUE;  switch(msg)  {  case WM\_CLOSE:  EndDialog(hdlg,FALSE);  break;  case WM\_COMMAND:  switch(LOWORD(wParam))  {  case IDOK:  if(IDOK==MessageBox(hdlg,TEXT("退出"),TEXT("确认"),MB\_OKCANCEL))  {  EndDialog(hdlg,FALSE);  }  break;  default:  ret = FALSE;  }  break;  case WM\_INITDIALOG:  break;  default:  ret = FALSE;  }  return ret;  } |

### 测试一下,

|  |
| --- |
|  |

### 点击按钮会弹出确认对话框

|  |
| --- |
|  |

### 点击确定退出程序,点击x也会退出程序

## 5.我们给对话框程序添加初始化对话框消息处理代码,我们在这里使用3种写法来实现同一个了,封装3个函数,一个是LoadListBox\_Stack使用栈来实现,一个是LoadListBox\_Heap使用堆来实现,另外一个LoadListBox\_Heap32使用windows的HeapAlloc函数和HeapFree函数

|  |
| --- |
| #include<Windows.h>  #include"resource.h"  INT\_PTR CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);  //使用栈的函数  void LoadListBox\_Stack(HWND hList)  {  TCHAR szVar[1024];//保存环境变量名称的数组  TCHAR\* pvarblock;  TCHAR\* pvarStart;//指向环境变量的第一个字母  TCHAR\* pvarEnd;//指向环境变量的最后一个字母  int len;//保存环境变量名称的长度  pvarblock = GetEnvironmentStrings();  while(\*pvarblock)  {  //上面的字符串中有一些变量是以=开头的,他们不是环境变量我们响应过滤他们  if(\*pvarblock !='=')  {  pvarStart = pvarblock;//记录起点  while(\*pvarblock++ !='=');  pvarEnd = pvarblock - 1;//进入终点,注意当\*pvarblock=='='时,已经超出了变量范围,所以要减掉1  len = pvarEnd - pvarStart;//计算环境变量名称的长度  ZeroMemory(szVar,1024);//清空缓冲区  //把环境变量从环境变量块里面提取出来  CopyMemory(szVar,pvarStart,len\*sizeof(TCHAR));  //添加到列表框,用SendMessage函数来实现  SendMessage(hList,LB\_ADDSTRING,0,(LPARAM)szVar);  }  while(\*pvarblock++ !='\0');//一直移动指针直到结束  }  pvarblock= GetEnvironmentStrings();//重新获取地址,方便释放内存;  FreeEnvironmentStrings(pvarblock);//释放内存之前需要把指针指向内存的收字节  }  //使用堆的c语言函数  void LoadListBox\_Heap(HWND hList)  {  //TCHAR szVar[1024];//保存环境变量名称的数组  TCHAR\* pvarName;  TCHAR\* pvarblock;  TCHAR\* pvarStart;//指向环境变量的第一个字母  TCHAR\* pvarEnd;//指向环境变量的最后一个字母  int len;//保存环境变量名称的长度  pvarblock = GetEnvironmentStrings();  while(\*pvarblock)  {  //上面的字符串中有一些变量是以=开头的,他们不是环境变量我们响应过滤他们  if(\*pvarblock !='=')  {  pvarStart = pvarblock;//记录起点  while(\*pvarblock++ !='=');  pvarEnd = pvarblock - 1;//进入终点,注意当\*pvarblock=='='时,已经超出了变量范围,所以要减掉1  len = pvarEnd - pvarStart;//计算环境变量名称的长度  //在堆中动态分配内存  pvarName = (TCHAR\*)calloc(len+1,sizeof(TCHAR));//ok的,它会在分配内存后进行初始化  //pvarName = (TCHAR\*)malloc((len+1) \* sizeof(TCHAR));//不好,分配内存后不初始化  //ZeroMemory(pvarName,(len+1) \* sizeof(TCHAR));//使用malloc分配到内存需要清空缓冲区  CopyMemory(pvarName,pvarStart,len\*sizeof(TCHAR));  //添加到列表框,用SendMessage函数来实现  SendMessage(hList,LB\_ADDSTRING,0,(LPARAM)pvarName);  free(pvarName);  }  while(\*pvarblock++ !='\0');//一直移动指针直到结束  }  pvarblock= GetEnvironmentStrings();//重新获取地址,方便释放内存;  FreeEnvironmentStrings(pvarblock);//释放内存之前需要把指针指向内存的收字节  }  //使用堆的Win32 API函数  void LoadListBox\_Heap32(HWND hList)  {    LPVOID lpVarName;  TCHAR\* pvarblock;  TCHAR\* pvarStart;//指向环境变量的第一个字母  TCHAR\* pvarEnd;//指向环境变量的最后一个字母  int len;//保存环境变量名称的长度  pvarblock = GetEnvironmentStrings();  while(\*pvarblock)  {  //上面的字符串中有一些变量是以=开头的,他们不是环境变量我们响应过滤他们  if(\*pvarblock !='=')  {  pvarStart = pvarblock;//记录起点  while(\*pvarblock++ !='=');  pvarEnd = pvarblock - 1;//进入终点,注意当\*pvarblock=='='时,已经超出了变量范围,所以要减掉1  len = pvarEnd - pvarStart;//计算环境变量名称的长度  //在堆中动态分配内存  lpVarName = HeapAlloc(GetProcessHeap(),0,len);//和malloc一样,分配内存后不会初始化  ZeroMemory(lpVarName,(len+1) \* sizeof(TCHAR));//初始化内存缓冲区  CopyMemory(lpVarName,pvarStart,len\*sizeof(TCHAR));  //添加到列表框,用SendMessage函数来实现  SendMessage(hList,LB\_ADDSTRING,0,(LPARAM)lpVarName);  HeapFree(GetProcessHeap(),0,lpVarName);  }  while(\*pvarblock++ !='\0');//一直移动指针直到结束  }  pvarblock= GetEnvironmentStrings();//重新获取地址,方便释放内存;  FreeEnvironmentStrings(pvarblock);//释放内存之前需要把指针指向内存的收字节  }  int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE prev,LPTSTR lpCmd,int nShow)  {  DialogBox(hInstance,MAKEINTRESOURCE(IDD\_DIALOG1),NULL,DlgProc);  return 0;  }  INT\_PTR CALLBACK DlgProc(HWND hdlg,UINT msg,WPARAM wParam,LPARAM lParam)  {  BOOL ret = TRUE;  switch(msg)  {  case WM\_CLOSE:  EndDialog(hdlg,FALSE);  break;  case WM\_INITDIALOG:  //LoadListBox\_Stack(GetDlgItem(hdlg,IDC\_LIST1));  //LoadListBox\_Heap(GetDlgItem(hdlg,IDC\_LIST1));  LoadListBox\_Heap32(GetDlgItem(hdlg,IDC\_LIST1));  break;  case WM\_COMMAND:  switch(LOWORD(wParam))  {  case IDOK:  if(IDOK==MessageBox(hdlg,TEXT("退出"),TEXT("确认"),MB\_OKCANCEL))  {  EndDialog(hdlg,FALSE);  }  break;  default:  ret = FALSE;  }  break;  default:  ret = FALSE;  }  return ret;  } |

## 6.上面我们成功在对话框启动的时候把所有环境变量都加载到列表框里面,下面我们需要做的就是当用户点击某一个列表项时,把他的值显示在下面的静态文本控件中,为了方便编程,我们把静态文本改为编辑框

|  |
| --- |
|  |

## 为了使得列表框能够自动换行,我们做了下面的设置

|  |
| --- |
|  |

## 我们在WM\_COMMAND消息处理方式的switch大门口那么添加下面的代码

|  |
| --- |
| case WM\_COMMAND:  switch(LOWORD(wParam))  {  case IDOK:  if(IDOK==MessageBox(hdlg,TEXT("退出"),TEXT("确认"),MB\_OKCANCEL))  {  EndDialog(hdlg,FALSE);  }  break;  case IDC\_LIST1:    hlist = GetDlgItem(hdlg,IDC\_LIST1);  if(HIWORD(wParam)==LBN\_SELCHANGE)  {  index = SendMessage(hlist,LB\_GETCURSEL,0,0);//获取当前选中是项  len = SendMessage(hlist,LB\_GETTEXTLEN,index,0);//获取选中项的文本长度  pvarName =(LPTSTR)HeapAlloc(GetProcessHeap(),0,len+1);//根据获取到的长度  ZeroMemory(pvarName,len);  SendMessage(hlist,LB\_GETTEXT,index,(LPARAM)pvarName);  //  // MessageBox(hdlg,pvarName,TEXT("确认"),MB\_OK);//获取到了变量名称  //获取变量值  //TCHAR szValue[4096];  //ZeroMemory(szValue,4096);  //获取环境变量的值可以分3步,  //第1步,先获取这个环境变量的值的长度,  // 第2步根据长度在堆中分配内存  //第3步获取值到内存缓冲区  len = GetEnvironmentVariable(pvarName,NULL,0)+1;//这样子使用就是获取值的长度  pvarValue = (TCHAR\*)calloc(len,sizeof(TCHAR));  GetEnvironmentVariable(pvarName,pvarValue,len+1);  SetDlgItemText(hdlg,IDC\_EDIT\_RESULT,pvarValue);    free(pvarValue);//使用calloc和malloc申请的堆内存需要使用free来释放  HeapFree(GetProcessHeap(),0,pvarName);  }  break;  default:  break;  }  break; |

### 效果

|  |
| --- |
|  |

## 这一节的完整代码如下

### environ.cpp

|  |
| --- |
| #include<Windows.h>  #include"resource.h"  INT\_PTR CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);  //使用栈的函数  void LoadListBox\_Stack(HWND hList)  {  TCHAR szVar[1024];//保存环境变量名称的数组  TCHAR\* pvarblock;  TCHAR\* pvarStart;//指向环境变量的第一个字母  TCHAR\* pvarEnd;//指向环境变量的最后一个字母  int len;//保存环境变量名称的长度  pvarblock = GetEnvironmentStrings();  while(\*pvarblock)  {  //上面的字符串中有一些变量是以=开头的,他们不是环境变量我们响应过滤他们  if(\*pvarblock !='=')  {  pvarStart = pvarblock;//记录起点  while(\*pvarblock++ !='=');  pvarEnd = pvarblock - 1;//进入终点,注意当\*pvarblock=='='时,已经超出了变量范围,所以要减掉1  len = pvarEnd - pvarStart;//计算环境变量名称的长度  ZeroMemory(szVar,1024);//清空缓冲区  //把环境变量从环境变量块里面提取出来  CopyMemory(szVar,pvarStart,len\*sizeof(TCHAR));  //添加到列表框,用SendMessage函数来实现  SendMessage(hList,LB\_ADDSTRING,0,(LPARAM)szVar);  }  while(\*pvarblock++ !='\0');//一直移动指针直到结束  }  pvarblock= GetEnvironmentStrings();//重新获取地址,方便释放内存;  FreeEnvironmentStrings(pvarblock);//释放内存之前需要把指针指向内存的收字节  }  //使用堆的c语言函数  void LoadListBox\_Heap(HWND hList)  {  //TCHAR szVar[1024];//保存环境变量名称的数组  TCHAR\* pvarName;  TCHAR\* pvarblock;  TCHAR\* pvarStart;//指向环境变量的第一个字母  TCHAR\* pvarEnd;//指向环境变量的最后一个字母  int len;//保存环境变量名称的长度  pvarblock = GetEnvironmentStrings();  while(\*pvarblock)  {  //上面的字符串中有一些变量是以=开头的,他们不是环境变量我们响应过滤他们  if(\*pvarblock !='=')  {  pvarStart = pvarblock;//记录起点  while(\*pvarblock++ !='=');  pvarEnd = pvarblock - 1;//进入终点,注意当\*pvarblock=='='时,已经超出了变量范围,所以要减掉1  len = pvarEnd - pvarStart;//计算环境变量名称的长度  //在堆中动态分配内存  pvarName = (TCHAR\*)calloc(len+1,sizeof(TCHAR));//ok的,它会在分配内存后进行初始化  //pvarName = (TCHAR\*)malloc((len+1) \* sizeof(TCHAR));//不好,分配内存后不初始化  //ZeroMemory(pvarName,(len+1) \* sizeof(TCHAR));//使用malloc分配到内存需要清空缓冲区  CopyMemory(pvarName,pvarStart,len\*sizeof(TCHAR));  //添加到列表框,用SendMessage函数来实现  SendMessage(hList,LB\_ADDSTRING,0,(LPARAM)pvarName);  free(pvarName);  }  while(\*pvarblock++ !='\0');//一直移动指针直到结束  }  pvarblock= GetEnvironmentStrings();//重新获取地址,方便释放内存;  FreeEnvironmentStrings(pvarblock);//释放内存之前需要把指针指向内存的收字节  }  //使用堆的Win32 API函数  void LoadListBox\_Heap32(HWND hList)  {    LPVOID lpVarName;  TCHAR\* pvarblock;  TCHAR\* pvarStart;//指向环境变量的第一个字母  TCHAR\* pvarEnd;//指向环境变量的最后一个字母  int len;//保存环境变量名称的长度  pvarblock = GetEnvironmentStrings();    while(\*pvarblock)  {  //上面的字符串中有一些变量是以=开头的,他们不是环境变量我们响应过滤他们  if(\*pvarblock !='=')  {  pvarStart = pvarblock;//记录起点  while(\*pvarblock++ !='=');  pvarEnd = pvarblock - 1;//进入终点,注意当\*pvarblock=='='时,已经超出了变量范围,所以要减掉1  len = pvarEnd - pvarStart;//计算环境变量名称的长度  //在堆中动态分配内存  lpVarName = HeapAlloc(GetProcessHeap(),0,len);//和malloc一样,分配内存后不会初始化  ZeroMemory(lpVarName,(len+1) \* sizeof(TCHAR));//初始化内存缓冲区  CopyMemory(lpVarName,pvarStart,len\*sizeof(TCHAR));  //添加到列表框,用SendMessage函数来实现  SendMessage(hList,LB\_ADDSTRING,0,(LPARAM)lpVarName);  HeapFree(GetProcessHeap(),0,lpVarName);  }  while(\*pvarblock++ !='\0');//一直移动指针直到结束  }  pvarblock = GetEnvironmentStrings();//重新获取地址,方便释放内存  FreeEnvironmentStrings(pvarblock);//释放内存之前需要把指针指向内存的收字节  }  int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE prev,LPTSTR lpCmd,int nShow)  {  DialogBox(hInstance,MAKEINTRESOURCE(IDD\_DIALOG1),NULL,DlgProc);  return 0;  }  INT\_PTR CALLBACK DlgProc(HWND hdlg,UINT msg,WPARAM wParam,LPARAM lParam)  {  BOOL ret = TRUE;  int index,len;//选中列表项的索引和文本长度  TCHAR \* pvarName,\*pvarValue;//保存环境变量名称和环境变量值  HWND hlist;  switch(msg)  {  case WM\_CLOSE:  EndDialog(hdlg,FALSE);  break;  case WM\_INITDIALOG:  // LoadListBox\_Stack(GetDlgItem(hdlg,IDC\_LIST1));  //LoadListBox\_Heap(GetDlgItem(hdlg,IDC\_LIST1));  LoadListBox\_Heap32(GetDlgItem(hdlg,IDC\_LIST1));  break;  case WM\_COMMAND:  switch(LOWORD(wParam))  {  case IDOK:  if(IDOK==MessageBox(hdlg,TEXT("退出"),TEXT("确认"),MB\_OKCANCEL))  {  EndDialog(hdlg,FALSE);  }  break;  case IDC\_LIST1:    hlist = GetDlgItem(hdlg,IDC\_LIST1);  if(HIWORD(wParam)==LBN\_SELCHANGE)  {  index = SendMessage(hlist,LB\_GETCURSEL,0,0);//获取当前选中是项  len = SendMessage(hlist,LB\_GETTEXTLEN,index,0);//获取选中项的文本长度  pvarName =(LPTSTR)HeapAlloc(GetProcessHeap(),0,len+1);//根据获取到的长度  ZeroMemory(pvarName,len);  SendMessage(hlist,LB\_GETTEXT,index,(LPARAM)pvarName);  //  // MessageBox(hdlg,pvarName,TEXT("确认"),MB\_OK);//获取到了变量名称  //获取变量值  //TCHAR szValue[4096];  //ZeroMemory(szValue,4096);  //获取环境变量的值可以分3步,  //第1步,先获取这个环境变量的值的长度,  // 第2步根据长度在堆中分配内存  //第3步获取值到内存缓冲区  len = GetEnvironmentVariable(pvarName,NULL,0)+1;//这样子使用就是获取值的长度  pvarValue = (TCHAR\*)calloc(len,sizeof(TCHAR));  GetEnvironmentVariable(pvarName,pvarValue,len+1);  SetDlgItemText(hdlg,IDC\_EDIT\_RESULT,pvarValue);    free(pvarValue);//使用calloc和malloc申请的堆内存需要使用free来释放  HeapFree(GetProcessHeap(),0,pvarName);  }  break;  default:  break;  }  break;  default:  ret = FALSE;  }  return ret;  } |

# 总结:

## 1.编程的时候如果需要获取值,可以使用栈,它比较简单,没有内存泄漏的危险,但是为了能够保存所有内容,时候我们必须申请比较大的内存,这样子如果获取的值比较小,就会造成内存浪费.

## 2.使用堆的话,可以先获取值的长度,然后在堆中申请内存空间,不过需要我们手动释放内存,否则会造成内存泄漏.

## 3.其实我们也可以可以先获取值的长度,然后在栈中申请空间,这样子就既不浪费内存,又不会造成内存泄漏.当然,如果我们需要的内容在1MB以上,那么就只能使用堆内存了,注意用完后释放内存就可以了.

## 4.如果我们移动了指向内存的指针,在释放这个内存空间之前,需要把指针重新设置为指向内存的首地址,然后再释放内存,否则会引起程序不正常退出

## 5.malloc和calloc函数是在进程的默认堆上面分配内存,如果需要创建新的堆,使用他们是不行的,就只能使用Win32的堆函数.

## 6.calloc函数在分配完内存空间后会对他进行初始化,malloc不会,win32堆函数也不会初始化,这一点需要注意

## 7.GetProcessHeap函数可以获取进程的默认堆

# 扩展:

## 小技巧1,设置编辑框控件自动换行

|  |
| --- |
| 新建一个编辑框控件（Edit Control）,将其多行（Multiline）前面打勾（属性设置为True）,Auto HScroll前面的勾去掉（属性设置False）,这样就可以实现每一行填满后自动换行了。 |

## 小技巧2.如果需要在堆中分配缓冲区来保存获取到的环境变量的值,需要先调用一次GetEnvironmentVariable,第2个参数传递NULL,第三个参数写0,来获取值的长度,然后根据这个长度来分配缓冲区用一个字符指针来接收,然后再调用一次GetEnvironmentVariable,第2个参数是缓冲区,第三个参数是缓冲区的大小,也就是值的长度+1,这个技巧需要好好掌握.

# 这一节的学习到此为止