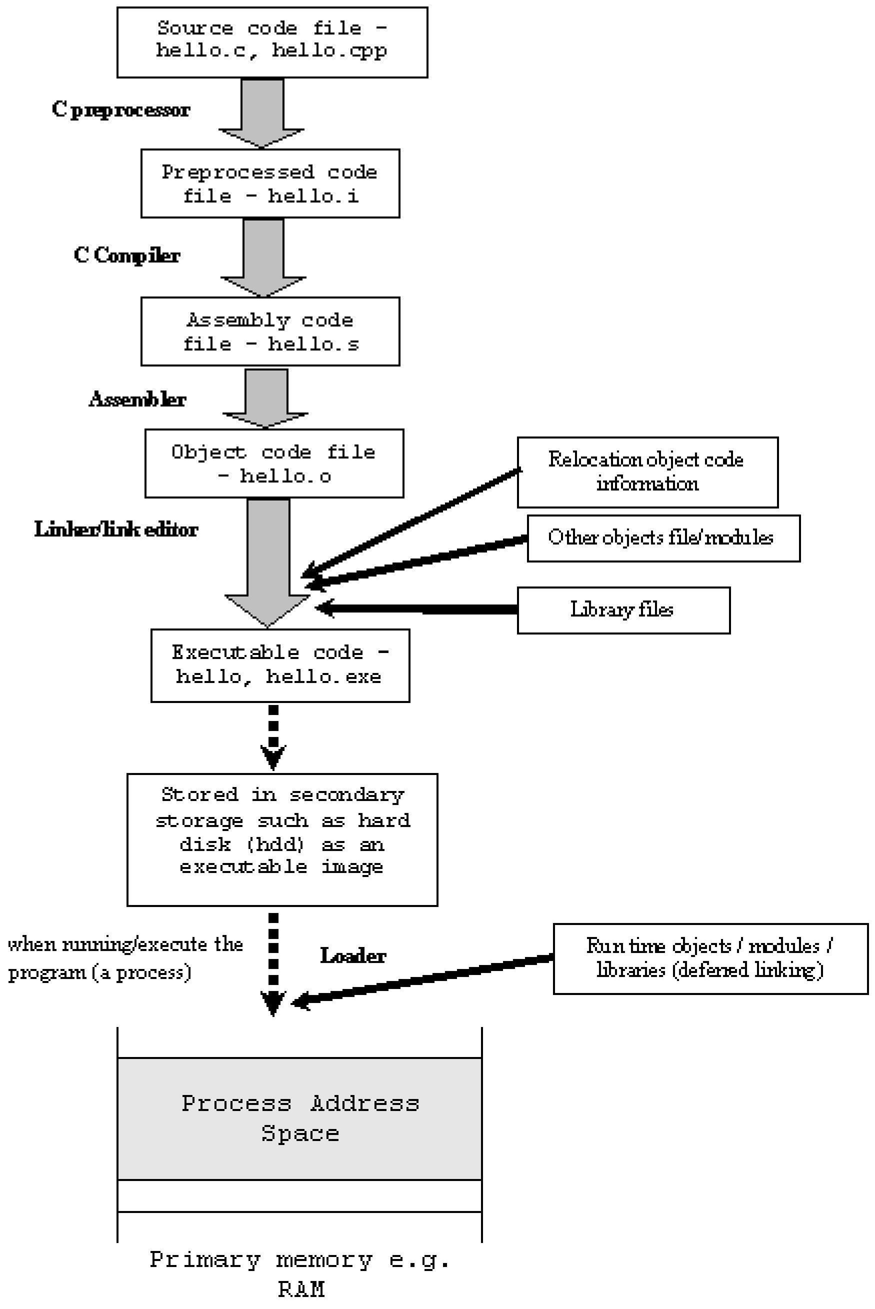
The Tool Help Library

What do we have in this session?

Introduction

The following Figure summarizes the steps involved in the process of building the C program starting from the compilation until the loading of the executable image into the memory for running the program.


Source code file - hello.c, hello.cpp
C preprocessor
Preprocessed code file - hello.i
C Compiler
Assembly code file - hello.s
Assembler
Object code file - hello.o
Relocation object code information
Other objects file/modules
Library files
Linker/link editor
Executable code - hello, hello.exe
Stored in secondary storage such as hard disk (hdd) as an executable image
when running/execute the program (a process)
Loader
Run time objects / modules / libraries (deferred linking)
Process Address Space
Primary memory e.g. RAM

Compile, link and execute stages for a running program

The Process (image)

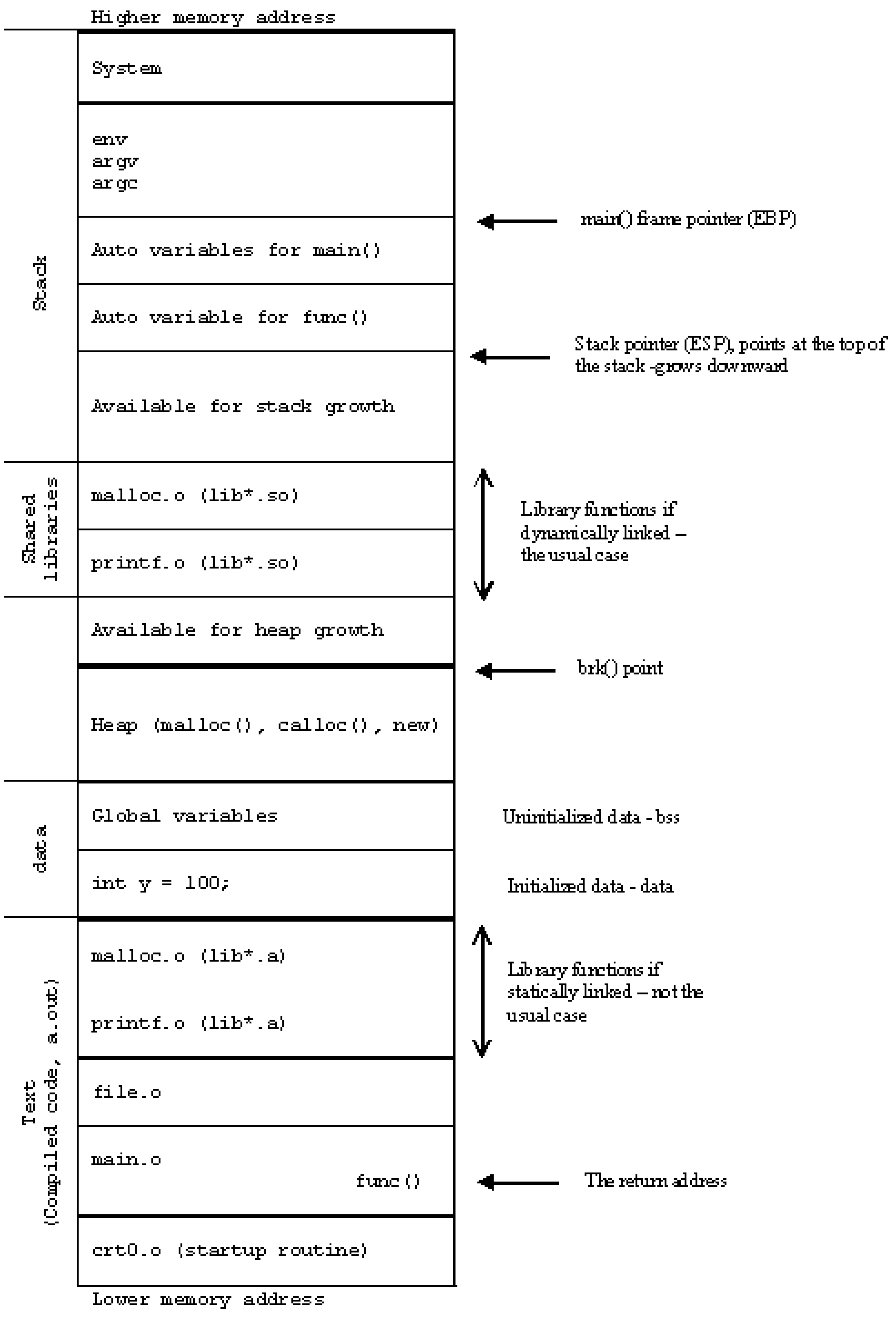
The diagram below shows the memory layout of a typical C's process (Linux/Unix). The process load segments (corresponding to "text" and "data" in the diagram) at the process's base address. The main stack is located just below and grows downwards. Any additional threads or function calls that are created will have their own stacks, located below the main stack.

Each of the stack frames is separated by a guard page to detect stack overflows among stacks frame. The heap is located above the process and grows upwards.

In the middle of the process's address space, there is a region is reserved for shared objects. When a new process is created, the process manager first maps the two segments from the executable into memory.

It then decodes the program's ELF header. If the program header indicates that the executable was linked against a shared library, the process manager will extract the name of the dynamic interpreter from the program header.

The dynamic interpreter points to a shared library that contains the runtime linker code. The process manager will load this shared library in memory and will then pass control to the runtime linker code in this library.


Higher memory address
System
env
argv
argc
Auto variables for main()
Auto variable for func()
Available for stack growth
malloc.o (lib*.so)
printf.o (lib*.so)
Available for heap growth
Heap (malloc(), calloc(), new)
Global variables
int y = 100;
malloc.o (lib*.a)
printf.o (lib*.a)
file.o
main.o
func()
crt0.o (startup routine)
Lower memory address
Stack
Shared libraries
data
Text (Compiled code, a.out)
main() frame pointer (EBP)
Stack pointer (ESP), points at the top of the stack -grows downward
Library functions if dynamically linked – the usual case
brk() point
Uninitialized data - bss
Initialized data - data
Library functions if statically linked – not the usual case
The return address

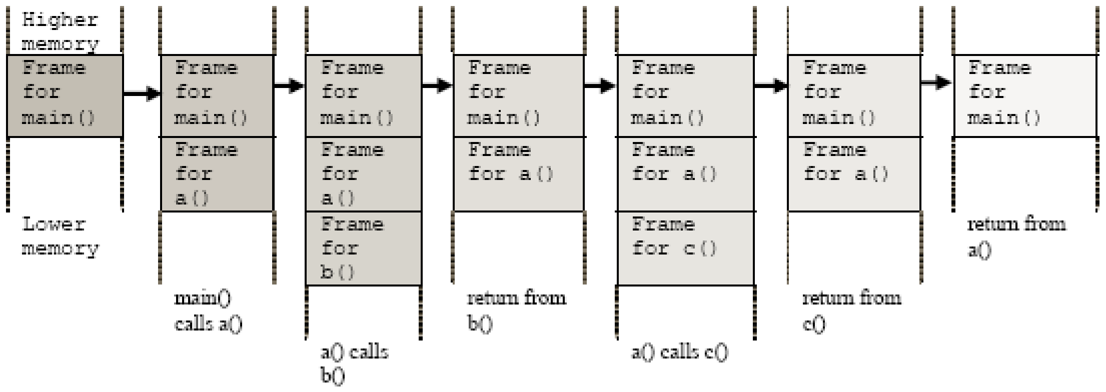
C’s process memory layout on an x86.

The Stack

Stack segment contains a stack, one entry/out, LIFO structure. On the x86 architecture, stacks grow downward, meaning that newer data will be allocated at addresses less than elements pushed onto the stack earlier. This stack normally called a stack frame (or a procedure activation record in java) when there are the boundaries pointed by EBP at the bottom and ESP at the top of the stack. Each function call creates a new stack frame and "stacked down" onto the previous stack(s), each one keeps track of the call chain or sequence that is which routine called it and where to return to, once it's done. Using a very simple C program skeleton, the following tries to figure out function calls and stack frames construction/destruction.

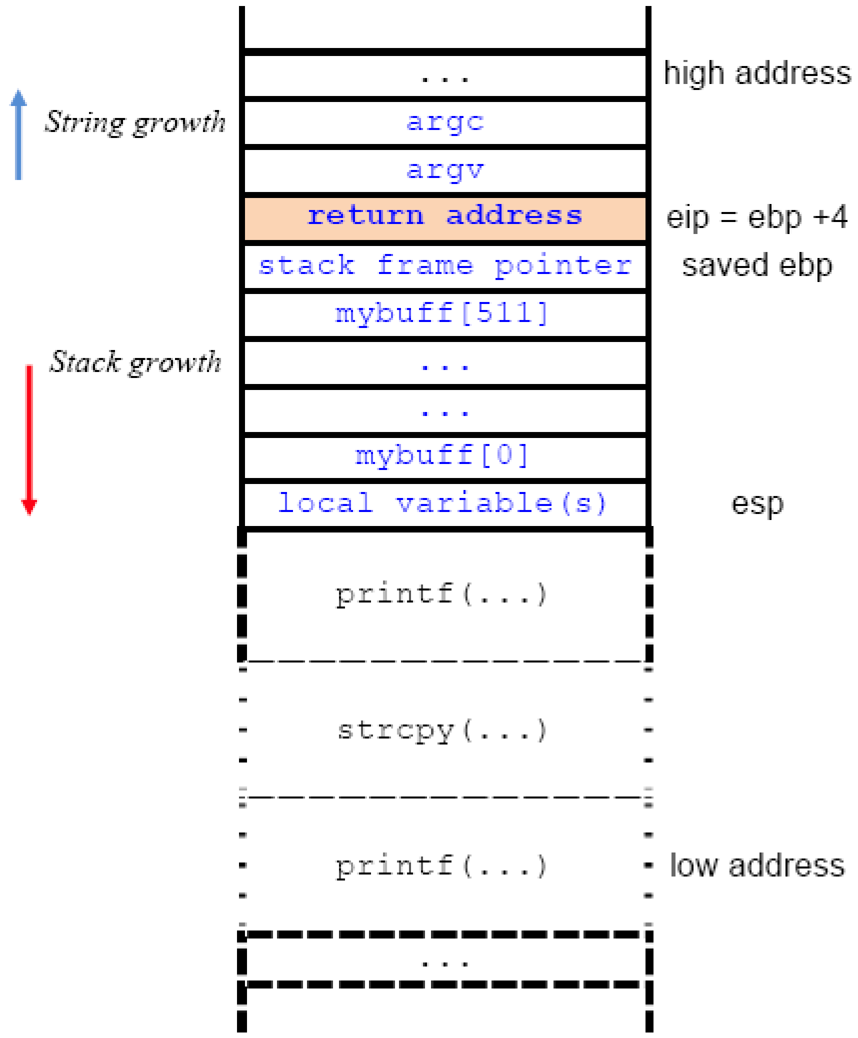
```
#include <stdio.h>

int a();
int b();
int c();

int a()
{
  b();
  c();
  return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }

int main()
{
  a();
  return 0;
}
```

By taking the stack area only, the following is what happen when the above program is run. At the end there should be equilibrium.



Stack frame and function call.

When a program begins execution in the function main(), stack frame is created, space is allocated on the stack for all variables declared within main(). Then, when main() calls a function, a(), new stack frame is created for the variables in a() at the top of the main() stack. Any parameters passed by main() to a() are stored on the stack. If a() were to call any additional functions such as b() and c(), new stack frames would be allocated at the new top of the stack. Notice that the order of the execution happened in the sequence. When c(), b() and a() return, storage for their local variables are de-allocated, the stack frames are destroyed and the top of the stack returns to the previous condition. The order of the execution is in the reverse. As can be seen, the memory allocated in the stack area is used and reused during program execution. It should be clear that memory allocated in this area will contain garbage values left over from previous usage.
During the execution the stack frame may shrink and grow and after the function call completed, the return address will be used to return to the caller and program execution continues. Then, the stack frame will be destroyed, releasing the memory to the system for other use. A simple idea is, if the return address can be changed, the program execution flows also can be changed.



A typical stack frame layout for C function call

The functions provided by the tool help library make it easier for you to obtain information about currently executing applications. These functions are designed to streamline the creation of tools, specifically debuggers.

Snapshots of the System

Snapshots are at the core of the tool help functions. A snapshot is a read-only copy of the current state of one or more of the following lists that reside in system memory:

1. Processes
2. Threads
3. Modules, and
4. Heaps

Processes that use tool help functions access these lists from snapshots instead of directly from the operating system. The lists in system memory change when processes are started and ended, threads are created and destroyed, executable modules are loaded and unloaded from system memory, and heaps are created and destroyed. The use of information from a snapshot prevents inconsistencies. Otherwise, changes to a list could possibly cause a thread to incorrectly traverse the list or cause an access violation (a GP fault). For example, if an application traverses the thread list while other threads are created or terminated, information that the application is using to traverse the thread list might become outdated and could cause an error for the application traversing the list.
To take a snapshot of the system memory, use the CreateToolhelp32Snapshot() function. You can control the content of a snapshot by specifying one or more of the following values when calling this function:

1. TH32CS_SNAPHEAPLIST
2. TH32CS_SNAPMODULE
3. TH32CS_SNAPPROCESS
4. TH32CS_SNAPTHREAD

The TH32CS_SNAPHEAPLIST and TH32CS_SNAPMODULE values are process specific. When these values are specified, the heap and module lists of the specified process are included in the snapshot. If you specify zero as the process identifier, the current process is used. The TH32CS_SNAPTHREAD value always creates a system-wide snapshot even if a process identifier is passed to CreateToolhelp32Snapshot().
To enumerate the heap or module state for all processes, specify the TH32CS_SNAPALL value and the process identifier of the current process. Then, for each additional process in the snapshot, call CreateToolhelp32Snapshot() again, specifying its process identifier and the TH32CS_SNAPHEAPLIST or TH32CS_SNAPMODULE value.

You can retrieve an extended error status code for CreateToolhelp32Snapshot() by using the GetLastError() function.
When your process finishes using a snapshot, destroy it using the CloseHandle() function. If you do not destroy a snapshot, the process will leak memory until it exits, at which time the system reclaims the memory.
The snapshot handle acts like a file handle and is subject to the same rules regarding the processes and threads in which it can be used. To specify that the handle is inheritable, create the snapshot using the TH32CS_INHERIT value.

Process Walking

A snapshot that includes the process list contains information about each currently executing process. You can retrieve information for the first process in the list by using the Process32First() function. After retrieving the first process in the list, you can traverse the process list for subsequent entries by using the Process32Next() function. Process32First() and Process32Next() fill a PROCESSENTRY32 structure with information about a process in the snapshot.
You can retrieve an extended error status code for Process32First() and Process32Next() by using the GetLastError() function.
You can read the memory in a specific process into a buffer by using the Toolhelp32ReadProcessMemory() function (or the VirtualQueryEx() function).
The contents of the th32ProcessID() and th32ParentProcessID() members of PROCESSENTRY32 are process identifiers and can be used by any functions that require a process identifier.

Thread Walking

A snapshot that includes the thread list contains information about each thread of each currently executing process. You can retrieve information for the first thread in the list by using the Thread32First() function. After retrieving the first thread in the list, you can retrieve information for subsequent threads by using the Thread32Next() function. Thread32First() and Thread32Next() fill a THREADENTRY32 structure with information about individual threads in the snapshot.
You can enumerate the threads of a specific process by taking a snapshot that includes the threads and then by traversing the thread list, keeping information about the threads that have the same process identifier as the specified process.
You can retrieve an extended error status code for Thread32First() and Thread32Next() by using the GetLastError() function.
The contents of the th32ThreadID member of THREADENTRY32 is a thread identifier and can be used by any functions that require a thread identifier.

Module Walking

A snapshot that includes the module list for a specified process contains information about each module, executable file, or dynamic-link library (DLL), used by the specified process. You can

retrieve information for the first module in the list by using the Module32First() function. After retrieving the first module in the list, you can retrieve information for subsequent modules in the list by using the Module32Next() function. Module32First() and Module32Next() fill a MODULEENTRY32 structure with information about the module.
You can retrieve an extended error status code for Module32First() and Module32Next() by using the GetLastError() function.
The module identifier, which is specified in the th32ModuleID member of MODULEENTRY32, only has meaning in 16-bit Windows.

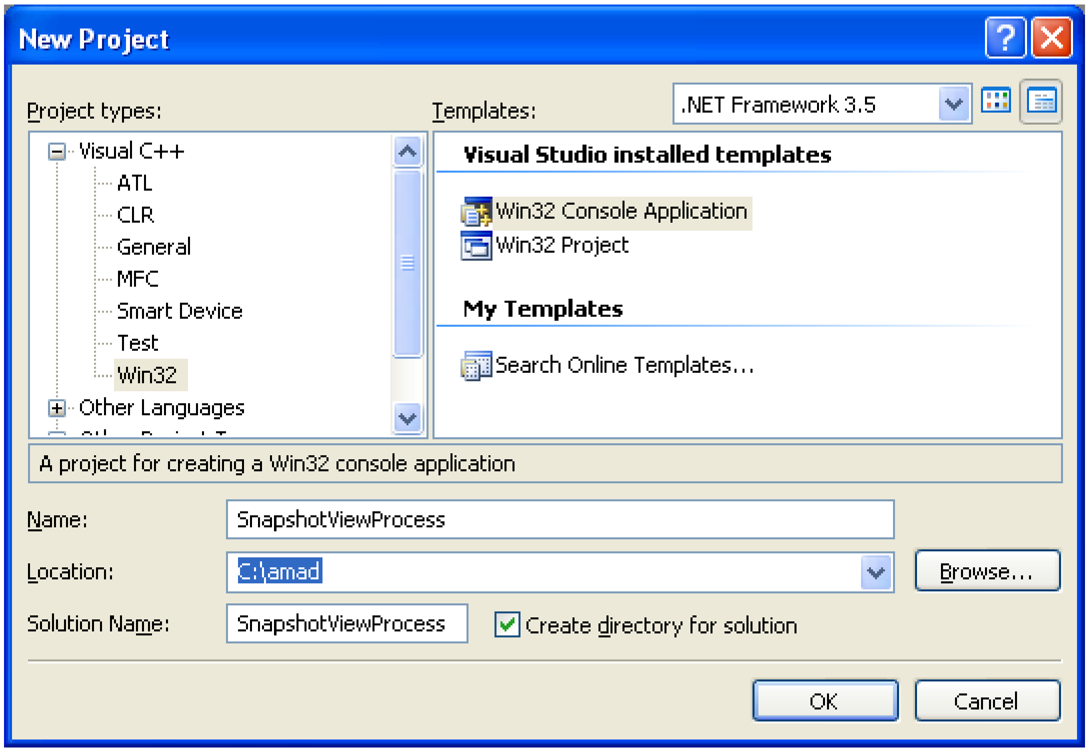
Heap Lists and Heap Walking

A snapshot that includes the heap list for a specified process contains identification information for each heap associated with the specified process and detailed information about each heap. You can retrieve an identifier for the first heap of the heap list by using the Heap32ListFirst() function. After retrieving the first heap in the list, you can traverse the heap list for subsequent heaps associated with the process by using the Heap32ListNext() function. Heap32ListFirst() and Heap32ListNext() fill a HEAPLIST32 structure with the process identifier, the heap identifier, and flags describing the heap.
You can retrieve information about the first block of a heap by using the Heap32First() function. After retrieving the first block of a heap, you can retrieve information about subsequent blocks of the same heap by using the Heap32Next() function. Heap32First() and Heap32Next() fill a HEAPENTRY32 structure with information for the appropriate block of a heap.
You can retrieve an extended error status code for Heap32ListFirst(), Heap32ListNext(), Heap32First(), and Heap32Next() by using the GetLastError() function.
The heap identifier, which is specified in the th32HeapID member of the HEAPENTRY32 structure, has meaning only to the tool help functions. It is not a handle, nor is it usable by other functions.

Using the Tool Help Functions: Program Examples

Taking a Snapshot and Viewing Processes Program Example

The following simple console application obtains a list of running processes. First, the GetProcessList() function takes a snapshot of currently executing processes in the system using CreateToolhelp32Snapshot(), and then it walks through the list recorded in the snapshot using Process32First() and Process32Next(). For each process in turn, GetProcessList() calls the ListProcessModules() function which is described in Traversing the Module List, and the ListProcessThreads() function which is described in Traversing the Thread List.
A simple error-reporting function, printError(), displays the reason for any failures, which usually result from security restrictions. For example, OpenProcess() fails for the Idle and CSRSS processes because their access restrictions prevent user-level code from opening them.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Project types: Visual C++ — ATL, CLR, General, MFC, Smart Device, Test, Win32; Other Languages

Templates: .NET Framework 3.5

Visual Studio installed templates: Win32 Console Application, Win32 Project

My Templates: Search Online Templates...

A project for creating a Win32 console application

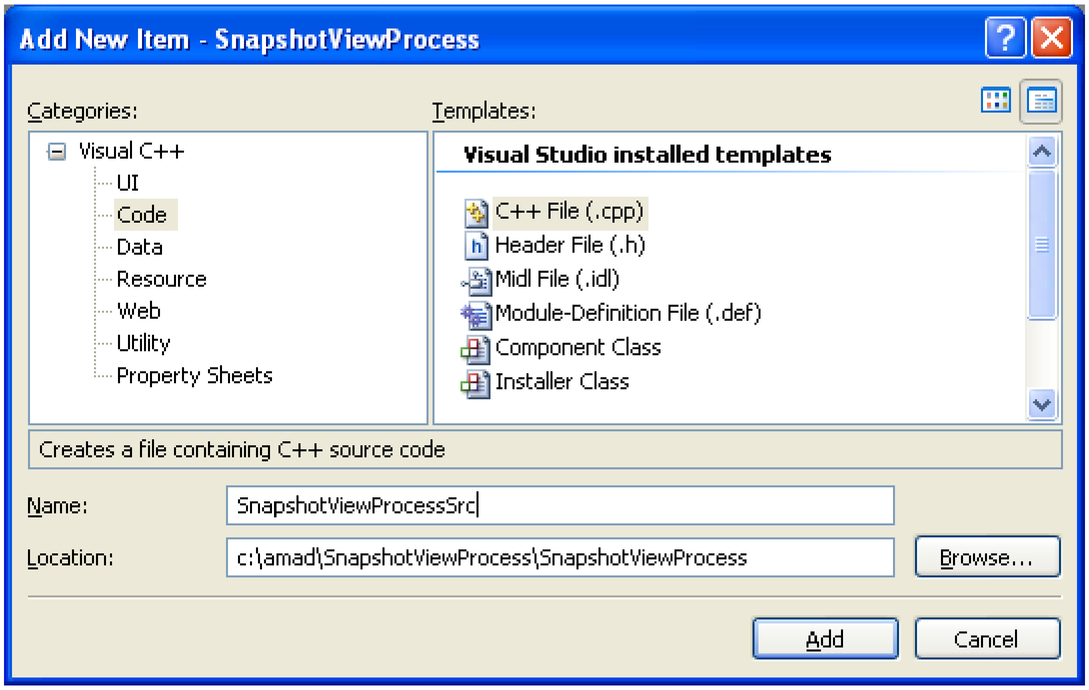
Name: SnapshotViewProcess

Location: C:\amad

Solution Name: SnapshotViewProcess

[✓] Create directory for solution

Then, add the source file and give it a suitable name.



Categories: Visual C++ — UI, Code, Data, Resource, Web, Utility, Property Sheets

Templates:

Visual Studio installed templates: C++ File (.cpp), Header File (.h), Midl File (.idl), Module-Definition File (.def), Component Class, Installer Class

Creates a file containing C++ source code

Name: SnapshotViewProcessSrc

Location: c:\amad\SnapshotViewProcess\SnapshotViewProcess

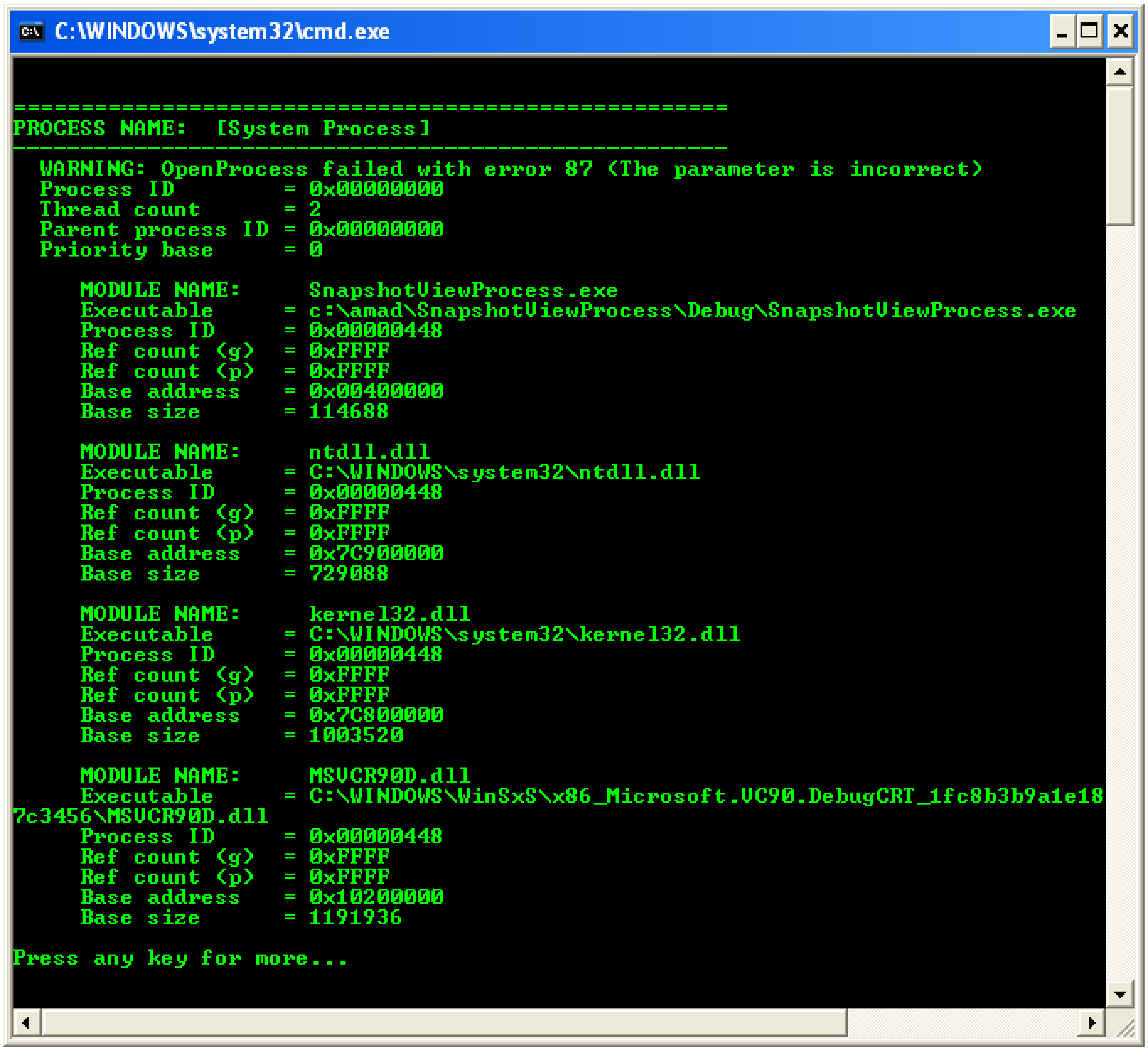
Next, add the following source code.

```
#include <windows.h>
```

```
#include <tlhelp32.h>
#include <stdio.h>

//  Forward declarations...
BOOL GetProcessList();
BOOL ListProcessModules(DWORD dwPID);
BOOL ListProcessThreads(DWORD dwOwnerPID);
void printError(WCHAR *msg );

int wmain()
{
  GetProcessList();
  return (TRUE);
}

BOOL GetProcessList()
{
  HANDLE hProcessSnap;
  HANDLE hProcess;
  PROCESSENTRY32 pe32;
  DWORD dwPriorityClass;

  // Take a snapshot of all processes in the system.
  hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

  if(hProcessSnap == INVALID_HANDLE_VALUE)
  {
    printError(L"CreateToolhelp32Snapshot (of processes)");
    return(FALSE);
  }

  // Set the size of the structure before using it.
  pe32.dwSize = sizeof(PROCESSENTRY32);

  // Retrieve information about the first process,
  // and exit if unsuccessful
  if(!Process32First( hProcessSnap, &pe32))
  {
    printError(L"Process32First"); // show cause of failure
    CloseHandle(hProcessSnap);     // clean the snapshot object
    return(FALSE);
  }

  // Now walk the snapshot of processes, and
  // display information about each process in turn
  do
  {
    wprintf(L"\n\n=====================================================");
    wprintf(L"\nPROCESS NAME:  %s", pe32.szExeFile);
    wprintf(L"\n---------------------------------------------------");

    // Retrieve the priority class.
    dwPriorityClass = 0;
    hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pe32.th32ProcessID);

    if(hProcess == NULL)
      printError(L"OpenProcess");
    else
```

```
      {
        dwPriorityClass = GetPriorityClass(hProcess);
        if(!dwPriorityClass)
          printError(L"GetPriorityClass");
        CloseHandle(hProcess);
      }

      wprintf(L"\n  Process ID        = 0x%08X", pe32.th32ProcessID);
      wprintf(L"\n  Thread count      = %d",   pe32.cntThreads);
      wprintf(L"\n  Parent process ID = 0x%08X", pe32.th32ParentProcessID);
      wprintf(L"\n  Priority base     = %d", pe32.pcPriClassBase);

      if(dwPriorityClass)
        wprintf(L"\n  Priority class    = %d", dwPriorityClass);

      // List the modules and threads associated with this process
      ListProcessModules(pe32.th32ProcessID);
      ListProcessThreads(pe32.th32ProcessID);
    } while(Process32Next(hProcessSnap, &pe32));

    CloseHandle(hProcessSnap);
    return(TRUE);
  }

  BOOL ListProcessModules(DWORD dwPID)
  {
    HANDLE hModuleSnap = INVALID_HANDLE_VALUE;
    MODULEENTRY32 me32;

    // Take a snapshot of all modules in the specified process.
    hModuleSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwPID);
    if(hModuleSnap == INVALID_HANDLE_VALUE)
    {
      printError(L"CreateToolhelp32Snapshot (of modules)");
      return(FALSE);
    }

    // Set the size of the structure before using it.
    me32.dwSize = sizeof(MODULEENTRY32);

    // Retrieve information about the first module,
    // and exit if unsuccessful
    if(!Module32First(hModuleSnap, &me32))
    {
      printError(L"Module32First");  // show cause of failure
      CloseHandle(hModuleSnap);      // clean the snapshot object
      return(FALSE);
    }

    // Now walk the module list of the process,
    // and display information about each module
    do
    {
      wprintf( L"\n\n     MODULE NAME:     %s",   me32.szModule );
      wprintf( L"\n     Executable     = %s",     me32.szExePath );
      wprintf( L"\n     Process ID     = 0x%08X",         me32.th32ProcessID );
      wprintf( L"\n     Ref count (g)  = 0x%04X",     me32.GlblcntUsage );
      wprintf( L"\n     Ref count (p)  = 0x%04X",     me32.ProccntUsage );
      wprintf( L"\n     Base address   = 0x%08X", (DWORD) me32.modBaseAddr );
      wprintf( L"\n     Base size      = %d",             me32.modBaseSize );
    } while(Module32Next(hModuleSnap, &me32));

    CloseHandle(hModuleSnap);
      wprintf(L"\n\nPress any key for more...\n");

    getwchar();
    return(TRUE);
  }

  BOOL ListProcessThreads(DWORD dwOwnerPID)
  {
    HANDLE hThreadSnap = INVALID_HANDLE_VALUE;
    THREADENTRY32 te32;

    // Take a snapshot of all running threads
    hThreadSnap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    if(hThreadSnap == INVALID_HANDLE_VALUE)
      return(FALSE);

    // Fill in the size of the structure before using it.
    te32.dwSize = sizeof(THREADENTRY32);

    // Retrieve information about the first thread, and exit if unsuccessful
    if(!Thread32First(hThreadSnap, &te32))
    {
      printError(TEXT("Thread32First")); // show cause of failure
      CloseHandle(hThreadSnap);          // clean the snapshot object
      return(FALSE);
    }

    // Now walk the thread list of the system,
    // and display information about each thread
    // associated with the specified process
    do
    {
      if(te32.th32OwnerProcessID == dwOwnerPID)
      {
        wprintf(L"\n\n     THREAD ID      = 0x%08X", te32.th32ThreadID);
        wprintf(L"\n     Base priority  = %d", te32.tpBasePri);
        wprintf(L"\n     Delta priority = %d", te32.tpDeltaPri);
      }
    } while(Thread32Next(hThreadSnap, &te32));

    CloseHandle(hThreadSnap);

    wprintf(L"\n\nPress any key for more...\n");
      getwchar();

    return(TRUE);
  }

  void printError(WCHAR * msg)
  {
    DWORD eNum;
    TCHAR sysMsg[256];
    TCHAR* p;

    eNum = GetLastError();

    FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
           NULL, eNum,
           MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
           sysMsg, 256, NULL);

    // Trim the end of the line and terminate it with a null
    p = sysMsg;
    while((*p > 31) || (*p == 9))
      ++p;
    do
    {
          *p-- = 0;
    } while((p >= sysMsg) && ((*p == '.') || (*p < 33)));

    // Display the message
    wprintf(L"\n  WARNING: %s failed with error %d (%s)", msg, eNum, sysMsg);
  }
```

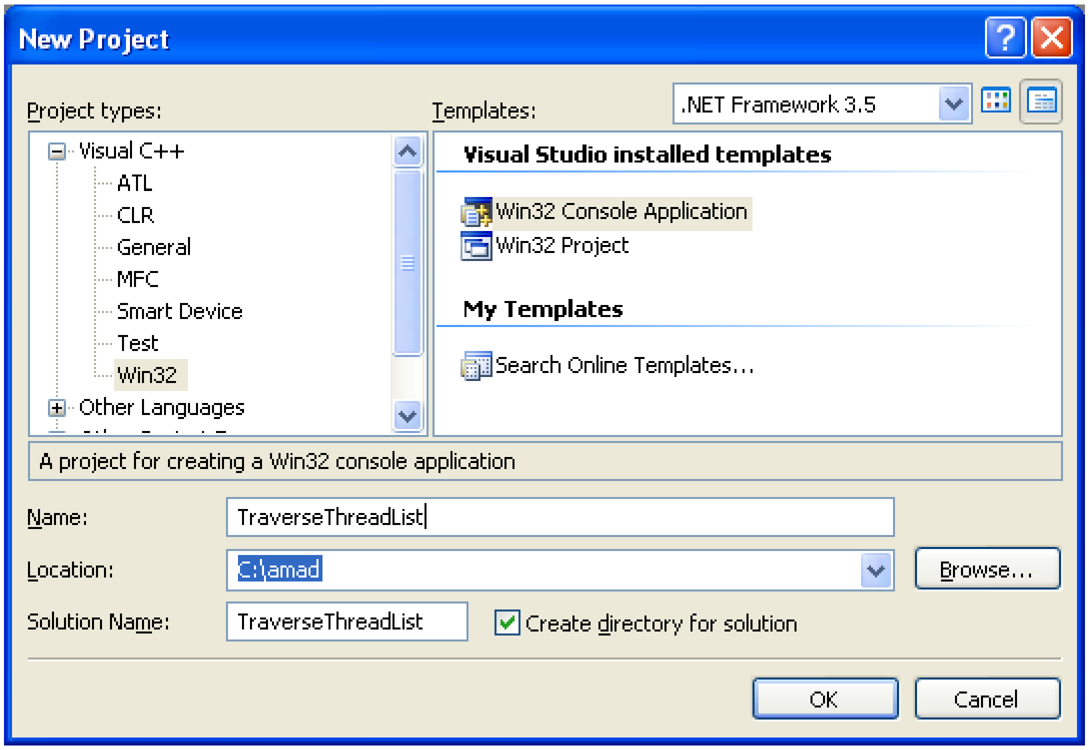
Build and run the project. The following screenshot is a sample output. Press any key for more output and Ctrl + C to terminate.



```
=====================================================
PROCESS NAME:  [System Process]
-----------------------------------------------------
  WARNING: OpenProcess failed with error 87 (The parameter is incorrect)
  Process ID        = 0x00000000
  Thread count      = 2
  Parent process ID = 0x00000000
  Priority base     = 0

     MODULE NAME:     SnapshotViewProcess.exe
     Executable     = c:\amad\SnapshotViewProcess\Debug\SnapshotViewProcess.exe
     Process ID     = 0x00000448
     Ref count (g)  = 0xFFFF
     Ref count (p)  = 0xFFFF
     Base address   = 0x00400000
     Base size      = 114688

     MODULE NAME:     ntdll.dll
     Executable     = C:\WINDOWS\system32\ntdll.dll
     Process ID     = 0x00000448
     Ref count (g)  = 0xFFFF
     Ref count (p)  = 0xFFFF
     Base address   = 0x7C900000
     Base size      = 729088

     MODULE NAME:     kernel32.dll
     Executable     = C:\WINDOWS\system32\kernel32.dll
     Process ID     = 0x00000448
     Ref count (g)  = 0xFFFF
     Ref count (p)  = 0xFFFF
     Base address   = 0x7C800000
     Base size      = 1003520

     MODULE NAME:     MSVCR90D.dll
     Executable     = C:\WINDOWS\WinSxS\x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18
7c3456\MSVCR90D.dll
     Process ID     = 0x00000448
     Ref count (g)  = 0xFFFF
     Ref count (p)  = 0xFFFF
     Base address   = 0x10200000
     Base size      = 1191936

Press any key for more...
```

Traversing the Thread List Program Example

The following example function lists running threads for a specified process. First, the ListProcessThreads() function takes a snapshot of the currently executing threads in the system using CreateToolhelp32Snapshot(), and then it walks through the list recorded in the snapshot using the Thread32First() and Thread32Next() functions. The parameter for ListProcessThreads() is the process identifier of the process whose threads are to be listed.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



.NET Framework 3.5

A project for creating a Win32 console application

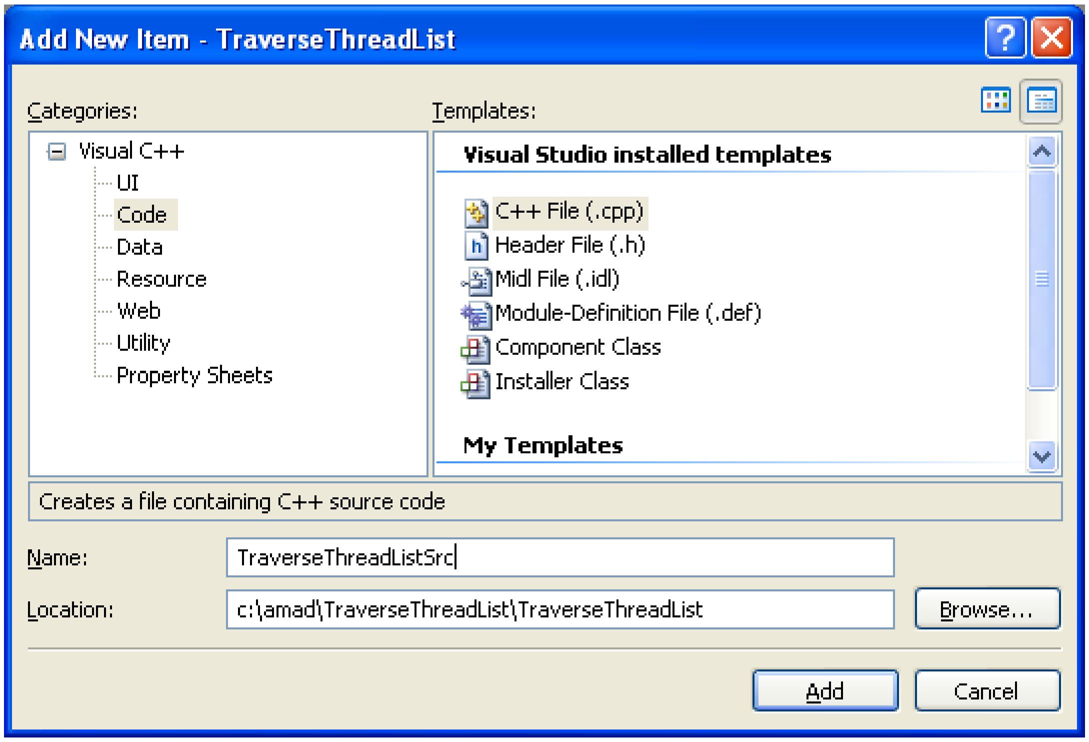
Name: TraverseThreadList

Location: C:\amad

Solution Name: TraverseThreadList

✓ Create directory for solution

Then, add the source file and give it a suitable name.



Creates a file containing C++ source code

Name: TraverseThreadListSrc

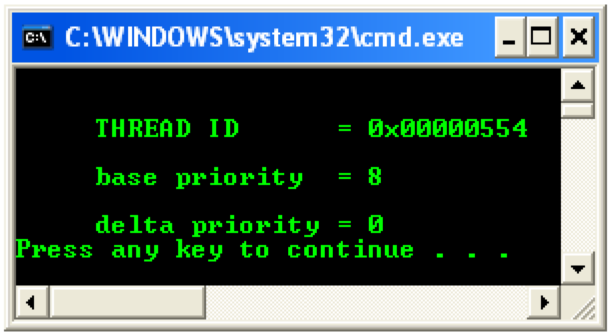
Location: c:\amad\TraverseThreadList\TraverseThreadList

Next, add the following source code.

```
#include <windows.h>
```

```c
#include <tlhelp32.h>
#include <stdio.h>

//  Forward declarations:
BOOL ListProcessThreads( DWORD dwOwnerPID);
void printError(WCHAR* msg);

int wmain()
{
  ListProcessThreads(GetCurrentProcessId());
}

BOOL ListProcessThreads(DWORD dwOwnerPID)
{
  HANDLE hThreadSnap = INVALID_HANDLE_VALUE;
  THREADENTRY32 te32;

  // Take a snapshot of all running threads
  hThreadSnap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
  if(hThreadSnap == INVALID_HANDLE_VALUE)
    return(FALSE);

  // Fill in the size of the structure before using it.
  te32.dwSize = sizeof(THREADENTRY32);

  // Retrieve information about the first thread,
  // and exit if unsuccessful
  if(!Thread32First(hThreadSnap, &te32))
  {
    printError(L"Thread32First\n"); // Show cause of failure
    CloseHandle(hThreadSnap);     // Must clean up the snapshot object!
    return(FALSE);
  }

  // Now walk the thread list of the system,
  // and display information about each thread
  // associated with the specified process
  do
  {
    if(te32.th32OwnerProcessID == dwOwnerPID)
    {
      wprintf(L"\n\n     THREAD ID      = 0x%08X\n", te32.th32ThreadID);
      wprintf(L"\n     base priority  = %d\n", te32.tpBasePri);
      wprintf(L"\n     delta priority = %d\n", te32.tpDeltaPri);
    }
  } while(Thread32Next(hThreadSnap, &te32));
  //  Don't forget to clean up the snapshot object.
  CloseHandle(hThreadSnap);
  return(TRUE);
}

void printError(WCHAR* msg)
{
  DWORD eNum;
  WCHAR sysMsg[256];
  WCHAR* p;

  eNum = GetLastError();
  FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
         NULL, eNum,
         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
         sysMsg, 256, NULL);

  // Trim the end of the line and terminate it with a null
  p = sysMsg;
  while((*p > 31) || (*p == 9))
    ++p;
  do {
        *p-- = 0;
  } while((p >= sysMsg) && ((*p == '.') || (*p < 33)));

  // Display the message
  wprintf(L"\n  WARNING: %s failed with error %d (%s)\n", msg, eNum, sysMsg);
}
```

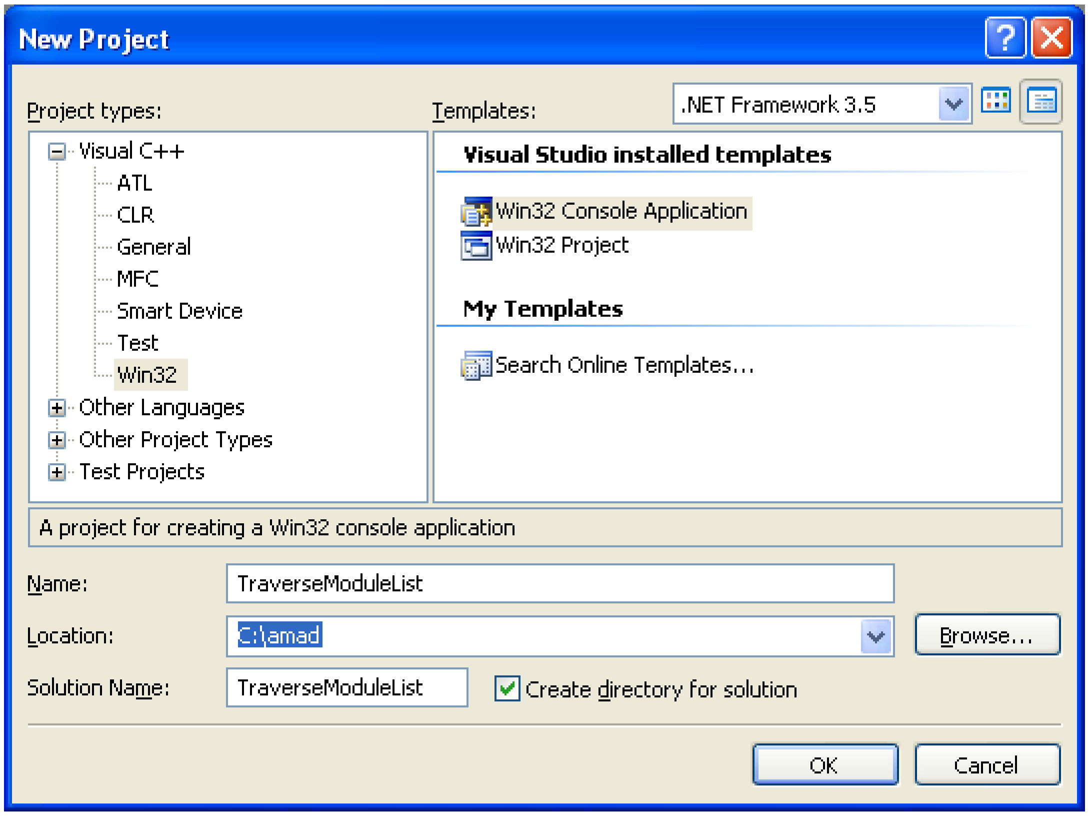
Build and run the project. The following screenshot is a sample output.



```

     THREAD ID       = 0x00000554

     base priority   = 8

     delta priority = 0
Press any key to continue . . .
```

Traversing the Module List Program Example

The following example obtains a list of modules for the specified process. The ListProcessModules() function takes a snapshot of the modules associated with a given process using the CreateToolhelp32Snapshot() function, and then walks through the list using the Module32First() and Module32Next() functions. The dwPID parameter of ListProcessModules() identifies the process for which modules are to be enumerated, and is usually obtained by calling CreateToolhelp32Snapshot() to enumerate the processes running on the system. See Taking a Snapshot and Viewing Processes for a simple console application that uses this function.
A simple error-reporting function, printError(), displays the reason for any failures, which usually result from security restrictions.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



A project for creating a Win32 console application

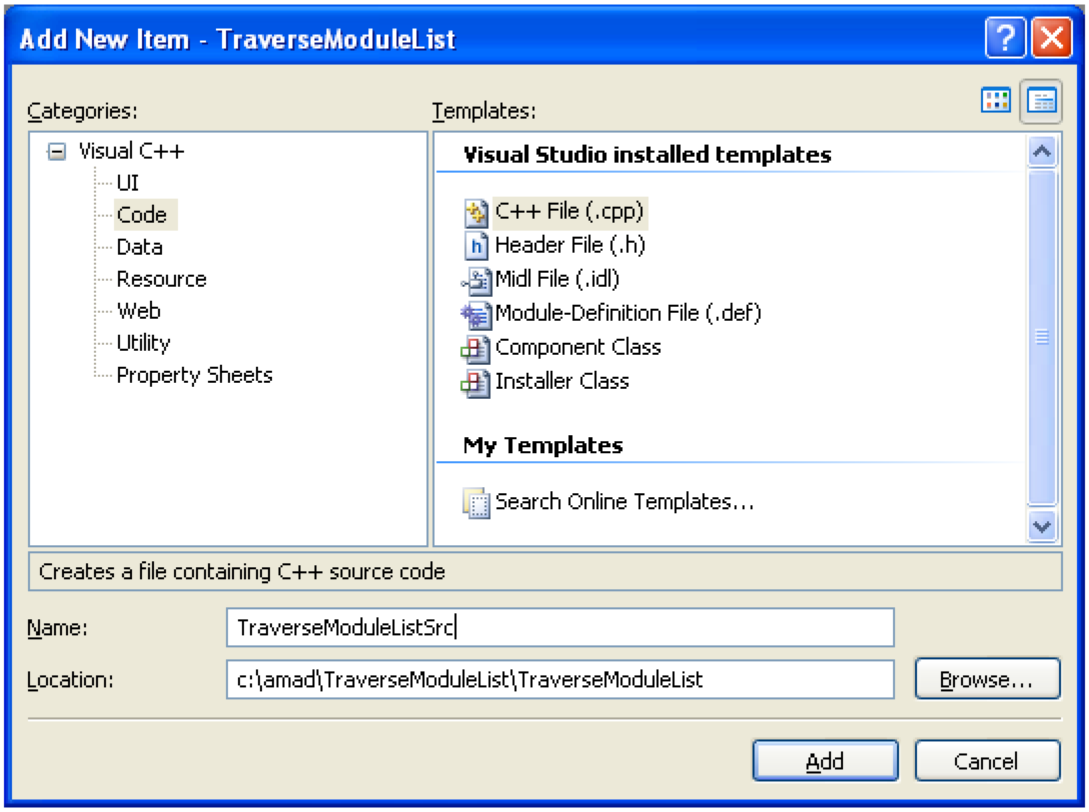
Name: TraverseModuleList

Location: C:\amad

Solution Name: TraverseModuleList

✓ Create directory for solution

Then, add the source file and give it a suitable name.



Creates a file containing C++ source code

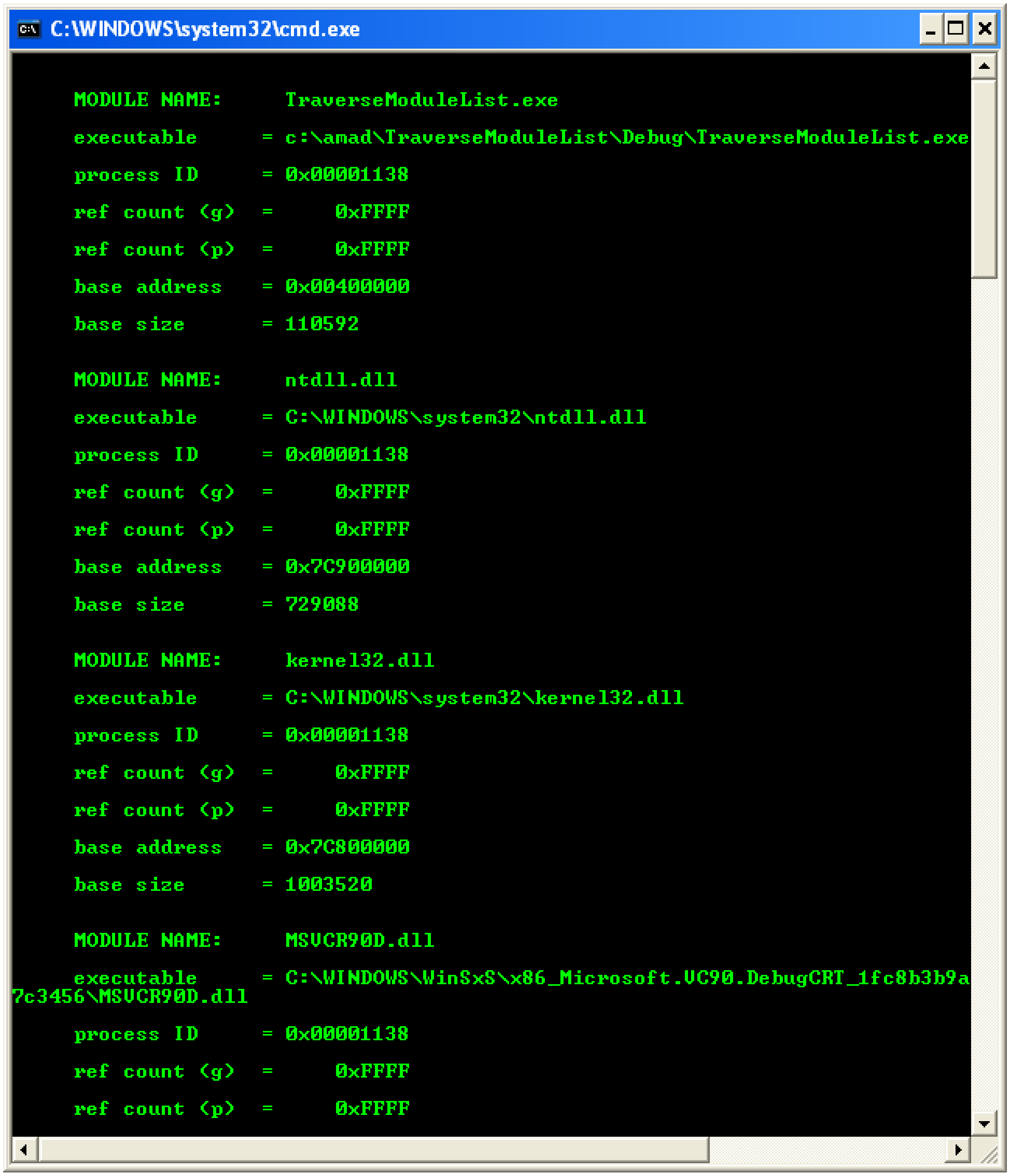
Name: TraverseModuleListSrc

Location: c:\amad\TraverseModuleList\TraverseModuleList

Next, add the following source code.

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

//  Forward declarations:
BOOL ListProcessModules(DWORD dwPID);
void printError(WCHAR* msg);

int wmain()
{
  ListProcessModules(GetCurrentProcessId());
}

BOOL ListProcessModules(DWORD dwPID)
{
  HANDLE hModuleSnap = INVALID_HANDLE_VALUE;
  MODULEENTRY32 me32;

//  Take a snapshot of all modules in the specified process.
  hModuleSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwPID);
  if(hModuleSnap == INVALID_HANDLE_VALUE)
  {
    printError(L"CreateToolhelp32Snapshot (of modules)\n");
    return(FALSE);
  }

//  Set the size of the structure before using it.
  me32.dwSize = sizeof(MODULEENTRY32);

//  Retrieve information about the first module,
//  and exit if unsuccessful
  if(!Module32First(hModuleSnap, &me32))
  {
    printError(L"Module32First\n");  // Show cause of failure
    CloseHandle(hModuleSnap);     // Must clean up the snapshot object!
    return( FALSE );
  }

//  Now walk the module list of the process,
//  and display information about each module
  do
  {
    wprintf(L"\n\n     MODULE NAME:     %s\n",             me32.szModule );
    wprintf(L"\n     executable     = %s\n",             me32.szExePath );
    wprintf(L"\n     process ID     = 0x%08X\n",         me32.th32ProcessID );
    wprintf(L"\n     ref count (g)  =     0x%04X\n",     me32.GlblcntUsage );
    wprintf(L"\n     ref count (p)  =     0x%04X\n",     me32.ProccntUsage );
    wprintf(L"\n     base address   = 0x%08X\n", (DWORD) me32.modBaseAddr );
    wprintf(L"\n     base size      = %d\n",             me32.modBaseSize );

  } while(Module32Next(hModuleSnap, &me32));

//  Do not forget to clean up the snapshot object.
  CloseHandle(hModuleSnap);
  return(TRUE);
}

void printError(WCHAR* msg)
{
  DWORD eNum;
  TCHAR sysMsg[256];
  TCHAR* p;

  eNum = GetLastError();
  FormatMessage( FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
         NULL, eNum,
         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
         sysMsg, 256, NULL );

  // Trim the end of the line and terminate it with a null
  p = sysMsg;
  while((*p > 31) || (*p == 9))
    ++p;
  do {
        *p-- = 0;
  } while((p >= sysMsg) && ((*p == '.') || (*p < 33)));

  // Display the message
  wprintf(L"\n  WARNING: %s failed with error %d (%s)\n", msg, eNum, sysMsg );
}
```

Build and run the project. The following screenshot is a sample output.



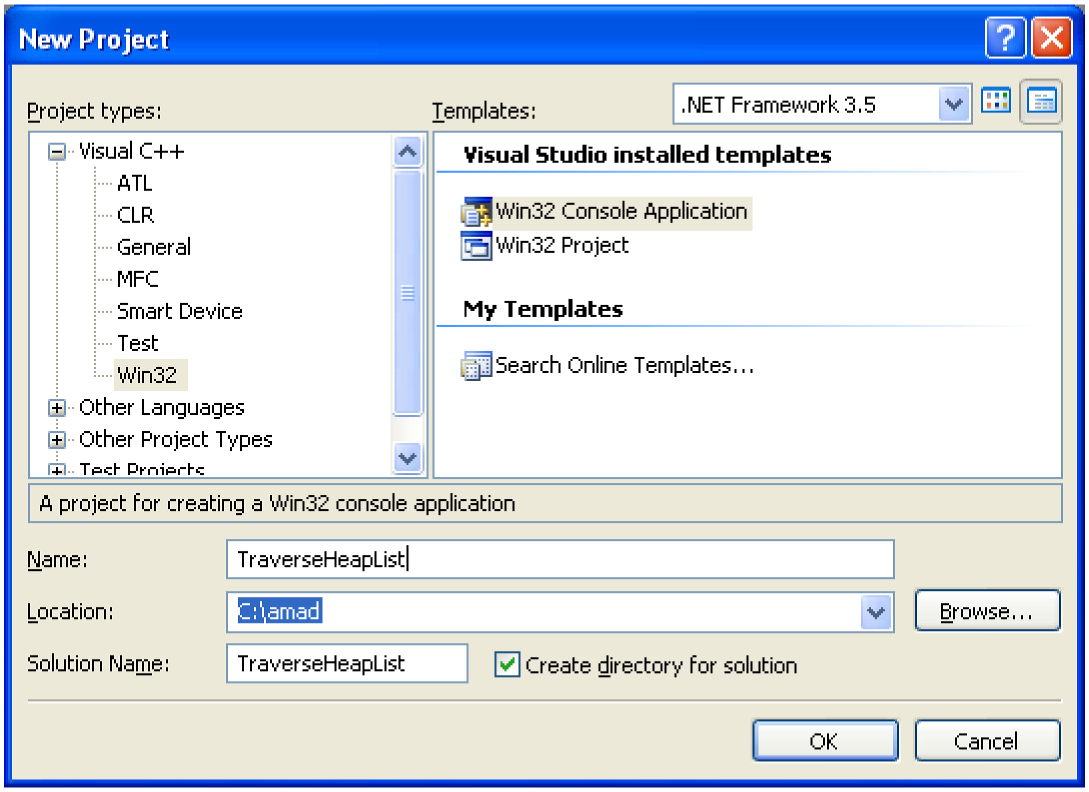
```
C:\WINDOWS\system32\cmd.exe

     MODULE NAME:     TraverseModuleList.exe
     executable      = c:\amad\TraverseModuleList\Debug\TraverseModuleList.exe
     process ID      = 0x00001138
     ref count (g)   =     0xFFFF
     ref count (p)   =     0xFFFF
     base address    = 0x00400000
     base size       = 110592

     MODULE NAME:     ntdll.dll
     executable      = C:\WINDOWS\system32\ntdll.dll
     process ID      = 0x00001138
     ref count (g)   =     0xFFFF
     ref count (p)   =     0xFFFF
     base address    = 0x7C900000
     base size       = 729088

     MODULE NAME:     kernel32.dll
     executable      = C:\WINDOWS\system32\kernel32.dll
     process ID      = 0x00001138
     ref count (g)   =     0xFFFF
     ref count (p)   =     0xFFFF
     base address    = 0x7C800000
     base size       = 1003520

     MODULE NAME:     MSVCR90D.dll
     executable      = C:\WINDOWS\WinSxS\x86_Microsoft.VC90.DebugCRT_1fc8b3b9a
7c3456\MSVCR90D.dll
     process ID      = 0x00001138
     ref count (g)   =     0xFFFF
     ref count (p)   =     0xFFFF
```

Traversing the Heap List Program Example

The following example obtains a list of heaps for the current process. It takes a snapshot of the heaps using the CreateToolhelp32Snapshot() function, and then walks through the list using the

Heap32ListFirst() and Heap32ListNext() functions. For each heap, it uses the Heap32First() and Heap32Next() functions to walk the heap blocks.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



A project for creating a Win32 console application

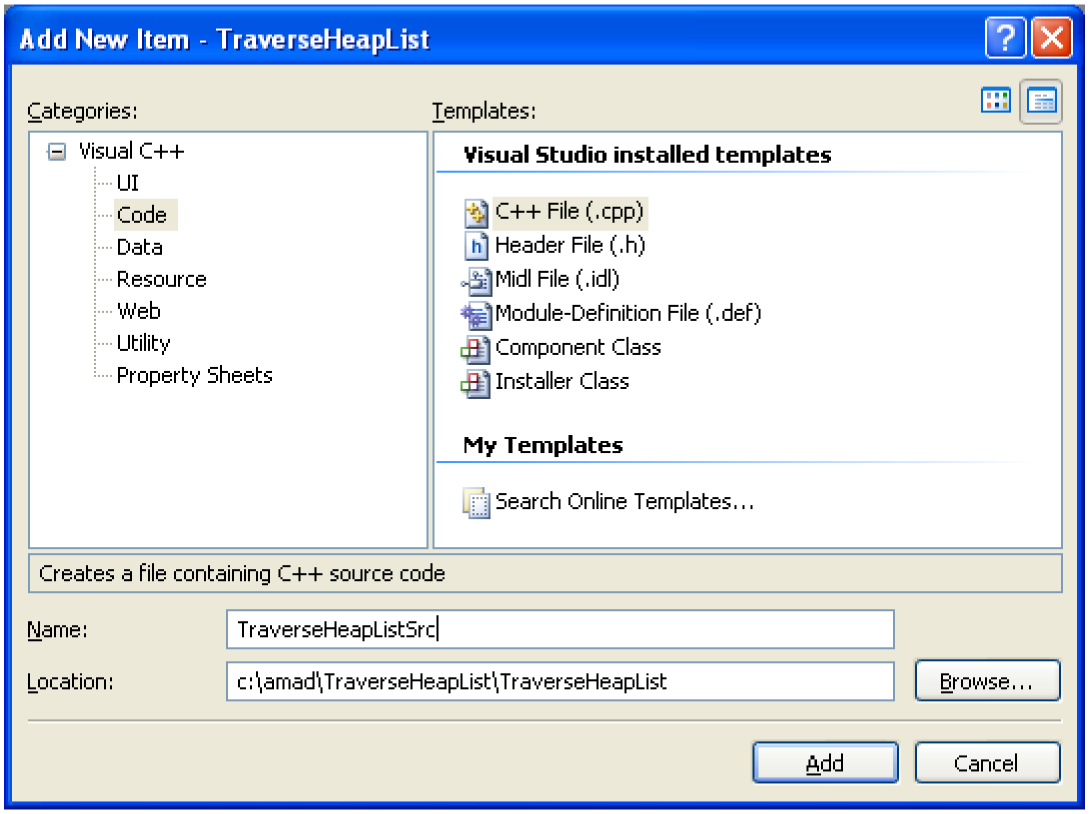
Name: TraverseHeapList

Location: C:\amad

Solution Name: TraverseHeapList

[✓] Create directory for solution

Then, add the source file and give it a suitable name.



Creates a file containing C++ source code

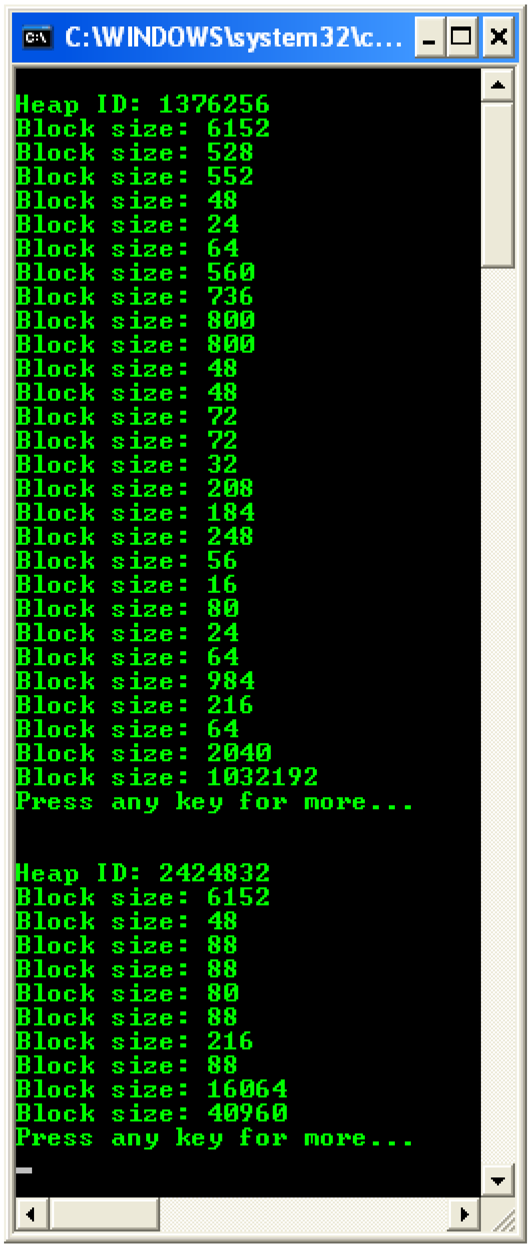
Name: TraverseHeapListSrc

Location: c:\amad\TraverseHeapList\TraverseHeapList

Next, add the following source code.

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

int main(int argc, WCHAR *argv[])
{
   HEAPLIST32 hl;

   HANDLE hHeapSnap = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST,
GetCurrentProcessId());

   hl.dwSize = sizeof(HEAPLIST32);

   if ( hHeapSnap == INVALID_HANDLE_VALUE )
   {
      wprintf (L"CreateToolhelp32Snapshot failed (%d)\n", GetLastError());
      return 1;
   }

   if( Heap32ListFirst( hHeapSnap, &hl ) )
   {
      do
      {
         HEAPENTRY32 he;
         ZeroMemory(&he, sizeof(HEAPENTRY32));
         he.dwSize = sizeof(HEAPENTRY32);
         if( Heap32First( &he, GetCurrentProcessId(), hl.th32HeapID ) )
         {
            wprintf(L"\nHeap ID: %d\n", hl.th32HeapID );
            do
            {
               wprintf(L"Block size: %d\n", he.dwBlockSize );

               he.dwSize = sizeof(HEAPENTRY32);
            } while( Heap32Next(&he) );
                  wprintf(L"Press any key for more...\n");
      getwchar();
         }
         hl.dwSize = sizeof(HEAPLIST32);
      } while (Heap32ListNext( hHeapSnap, &hl ));
                        wprintf(L"Press any key for more...\n");

      getwchar();
   }
   else
         wprintf(L"Cannot list first heap (%d)\n", GetLastError());

   CloseHandle(hHeapSnap);
}
```

Build and run the project. The following screenshot is a sample output.



```
Heap ID: 1376256
Block size: 6152
Block size: 528
Block size: 552
Block size: 48
Block size: 24
Block size: 64
Block size: 560
Block size: 736
Block size: 800
Block size: 800
Block size: 48
Block size: 48
Block size: 72
Block size: 72
Block size: 32
Block size: 208
Block size: 184
Block size: 248
Block size: 56
Block size: 16
Block size: 80
Block size: 24
Block size: 64
Block size: 984
Block size: 216
Block size: 64
Block size: 2040
Block size: 1032192
Press any key for more...


Heap ID: 2424832
Block size: 6152
Block size: 48
Block size: 88
Block size: 88
Block size: 80
Block size: 88
Block size: 216
Block size: 88
Block size: 16064
Block size: 40960
Press any key for more...
_
```

The Tool Help Reference

The following functions and structures are associated with the tool help library.

1. Tool Help Functions
2. Tool Help Structures