Chapter 6: The Linker

References:

- Brian W. Kernighan / Dennis M. Ritchie:
  The C Programming Language, 2nd Ed.
  Prentice-Hall, 1988.
- Samuel P. Harbison / Guy L. Steele Jr.:
  C — A Reference Manual, 4th Ed.
  Prentice-Hall, 1995.
- Online Documentation of Microsoft Visual C++ 6.0 (Standard Edition):
  MSDN Library: Visual Studio 6.0 release.
- Horst Wettstein: Assemblierer und Binder (in German).
  Carl Hanser Verlag, 1979.
- Peter Rechenberg, Gustav Pomberger (Eds.):
  Informatik-Handbuch (in German).
  Carl Hanser Verlag, 1997.
  Kapitel 12: Systemsoftware (H. Mössenböck).

Overview

Introduction (1)

- Often, a program consists of several modules which are separately compiled. Reasons are:
  - ◇ The program is large.

    Even with fast computers, editing and compiling a single file with a million lines leads to unnecessary delays.
  - ◇ The program is developed by several people.

    Different programmers cannot easily edit the same file at the same time. (There is software for collaborative work that permits that, but it is still a research topic.)
  - ◇ A large program is easier to understand if it is divided into natural units.

    E.g. each module defines one data type with its operations.

Introduction (2)

- Reasons for splitting a program into several source files (continued):
  - ◇ The same module might be used in different programs (e.g. library functions, code reuse).
  - ◇ Moduls might be written in different programming languages, but used in the same program.

    E.g. certain functions are coded in assembler (very performance-critical functions, functions that directly access the hardware). With C, this is seldom necessary. However, there might also be legacy code (old modules) that must be integrated into a new program.

Introduction (3)

- In C, separate compilation is the module mechanism that restricts the visibility of symbols on a higher level than single procedures:
  - ◇ Static functions/variables are global within a module, but not accessible from other modules.
  - ◇ Extern functions/variables are exported, and can be accessed in other modules.
- Types and macros are defined only while a single module is compiled, but they can be written into header files which are included by several modules.

Introduction (4)

- A C program nearly never defines all functions that it calls:
  - ◇ At least, it calls functions like `printf` that are defined in a library.
  - ◇ Such functions must be declared (e.g. by including `stdio.h`), so that the compiler can check parameter and return types.
  - ◇ However, these functions are not defined, i.e. no function body is given, not even in `stdio.h`.

Introduction (5)

- Therefore, the compiler does not directly produce an executable program. It produces an object file (a file with object code).
- The object file contains machine code, but with holes (for the addresses of unknown functions). It is input to another program, the linker.
- The linker adds machine code for the needed functions from a library (the standard C library). It puts the addresses of these functions in the places where they are called.

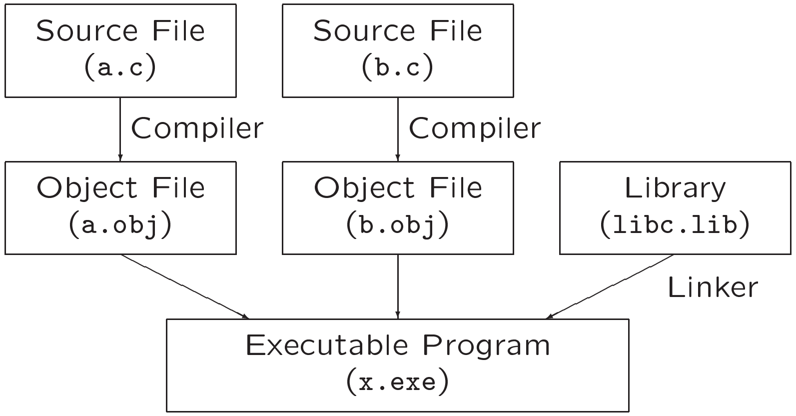
Introduction (6)

- As long as a project consists only of a single source file, the linker is automatically called after the compilation, so the differents might not be obvious.

  In MS VC++, the message window first shows "Compiling ..." and then "Linking ...". Object files produced by the compiler are stored in the configuration (version) directory, e.g. `Debug`.
  Under UNIX, one uses the option "`-c`" if one wants only to compile, e.g. `gcc -c a.c`. The result is then stored in `a.o`. It is easiest to call the linker (`ld`) also via `gcc`, e.g. `gcc -o prog a.o` (`-o` specifies the name of the output file which is by default `a.out`). Then `gcc` calls `ld`, but automatically adds the standard C library and sets certain options.

- But it is possible (and indeed very common) to write programs that consist of several source files.

Introduction (7)



Example (1)

- This is a legal C program (e.g. `a.c`) that will be compiled (into `a.obj`) without error or warning:

```
(1)   #include <stdio.h>
(2)   int p(int n);
(3)
(4)   int main()
(5)   {  int i;
(6)      i = p(1);
(7)      printf("The result is: %d\n", i);
(8)      return 0;
(9)   }
```


Example (2)

- The compiler knows the parameter and result type of `p` (from line 2) when it compiles line 6.
- It compiles the procedure call as usual, only it leaves the address in the `CALL` machine instruction open.

  E.g. it writes the address 0 there, and notes in the object file that at this place the address of `p` has to be put (besides the machine code, the object file contains a table of used, but not yet defined symbols).
- But when one tries to link `a.obj` to an executable program, one gets the following error message:

  `a.obj: unresolved external symbol _p`

Example (3)

- For some (probably historic) reason, the C compiler prefixes an underscore “_” to every external symbol it writes to the object file.

  This is called “name decoration”. It is also used to distinguish different calling conventions. In C++, much more name decoration is done because different classes can have methods with the same name, and methods can be overloaded with different parameter types.
- If the declaration of `p` were missing, the compiler would have printed a warning, but still would have generated the object file `a.obj`.

  For historic reasons, it is legal to call in C an undeclared function. The compiler assumes the return type `int` and does not check parameters.

Example (4)

- The function `p` can be defined in another C source file, e.g. `b.c`:

```
(1)   int p(int n)
(2)   {
(3)       return n * 2;
(4)   }
```

- This file is compiled into the object file "`b.obj`".
- Then the linker creates an executable program out of `a.obj`, `b.obj` and the standard C library.

Example (5)

- If one tries to link only `b.obj`, one gets the following error message:

```
LIBCD.lib(crt0.obj):
    unresolved external symbol _main
```

- The procedure "`main`" is not really where the execution starts: The standard library contains a routine that interfaces with the operating system and then calls your procedure `main`.

  The real entry-point of the executable program is probably the procedure "`mainCRTStartup`" defined in the library module "`crt0.obj`". E.g. the command line arguments (`argc`, `argv`) must be set up.

Example (6)

- It is possible to declare different parameter and return types in `b.c`:

```
(1)   void p(char c, char d)
(2)   {
(3)       printf("c = %c, d = %c\n", c, d);
(4)   }
```

- The compiler will not notice any error, since it compiles `a.c` and `b.c` separately.

  I.e. the user calls the compiler two times in different program invocations. When the compiler translates `a.c`, it knows nothing about `b.c` (`b.c` does not even need to exist at that point) and vice versa.

Example (7)

- The linker also reports no error: It does not understand C types (the same linker also links Assembler programs, Fortran programs, etc.).

  The type informations is not even contained in the object file that is input to the linker (except possibly in debugging information).

- So the program compiles and links without any error, but will output strange results.

  The procedure `p` will interpret the integer on the stack as two characters, in MS VC++ it prints the characters with codes 1 and 0 (a smilie and a blank). The return value of `p` is whatever happens to remain in the register `EAX` (in my case the number 13).

Example (8)

- In order for the C compiler to detect such errors, one writes the declaration of all exported procedures (variables, types, etc.) into header files that are included in the defining and the referencing module.
- In the example, we create a header file `b.h` that contains the declaration of `p`:

```
int p(int n);
```

- Usually one also puts a comment in this file that explains what `p` is supposed to do.

Example (9)

- This header file is now included in `a.c` (instead of declaring `p` explicitly):

```
#include <stdio.h>
#include "b.h"

int main()
{  int i;
   i = p(1);
   printf("The result is: %d\n", i);
   return 0;
}
```


Example (10)

- The header file `b.h` is also included in `b.c` (in addition to defining `p`):

```
(1)   #include "b.h"
(2)   int p(int n)
(3)   {
(4)      return n * 2;
(5)   }
```

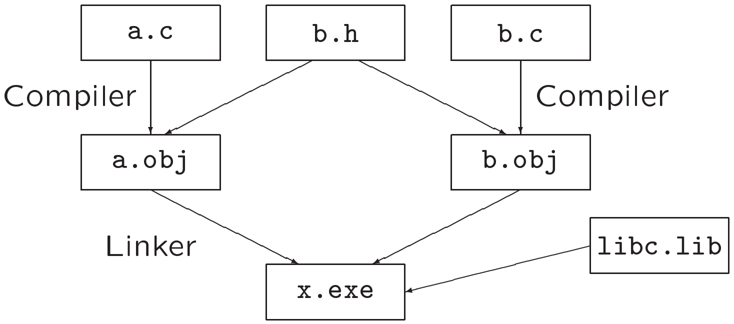
- If the result or parameter types in line 2 should differ from the declaration of `p` that the compiler has seen previously in `b.h`, it prints an error message.

Example (11)

- As often in C, a certain programming discipline (good programming style) is necessary to have a better protection against errors.
- One should avoid declaring the same function (variable, type) independently in two places: If later one of the two places is changed, the program becomes inconsistent.
- In old-style C, functions did not have to be declared when they are called. However, the program "`lint`" read all modules and checked them for consistency.

Example (12)

- The dependencies between the files can be depicted as follows:



Example (13)

- Splitting a large program into several modules also has the advantage that not every change requires a complete recompilation of all modules.
- In the example:
  - ◇ If `a.c` is changed, only `a.c` must be compiled.
  - ◇ If `b.c` is changed, only `b.c` must be compiled.
  - ◇ If `b.h` is changed, `a.c` and `b.c` must both be compiled, since they both include `b.h`.
- After compiling the affected modules, the linker must always be called.

Example (14)

- If there are more header files (and header files that themselves include other header files), it is quite difficult to decide which files need recompilation.
- Fortunately, the program development environment automatically keeps track of the dependencies.

  E.g. if one clicks on "Build x.exe" (F7) in VC++, it automatically compiles only those source files that need recompilation. If one fears that something went wrong (e.g. one adjusted the clock or copied object files from a floppy disk), one can click on "Rebuild All": This deletes all intermediate files and recompiles everything from scratch.
- Under UNIX, the program `make` performs this task.

Overview

1. Introduction (Overview)
2. Object Files, Libraries, and the Linker
3. Make
4. Dynamic Linking

Object Files (1)

- An object file basically contains:
  - ◇ Machine code for the defined functions (with holes for the addresses of unknown functions).
  - ◇ A table of defined symbols (functions, global variables) together with their value (address).
  - ◇ A table of used symbols together with the places where their value (address) must be inserted.
  - ◇ Information for code relocation (see below).
- However, besides program code, an object file contains also information about data (see next slide).

Object Files (2)

- The object file (and the executable program) is usually split into several sections/segments:
  - ◇ The "text" segment contains the machine instructions.
  - ◇ The "data" segment contains variables that are explicitly initialized.
  - ◇ The "bss" segment contains space for variables that will be implicitly set to 0.

    Whereas the executable program contains all initialized variables, it contains only the size of te memory that must be reserved for the uninitialized variables.

Object Files (3)

- Besides the standard data segment, there might also be a read-only data segment ("rdata").

  E.g. for string constants that appear in the program or for variables that are declared "`const`". The computer hardware might be able to protect the text segment (program code) and the read-only data segment from changes. If the same program is executed several times, the text segment and the read-only data segment can be shared.

- Finally, there may also be sections that contain debugging information and other information (e.g. the source filename, needed libraries, compiler options).

Object Files (4)

- Visual C++ comes with a program `dumpbin` that can be used to show the contents of an object file.

  It might be necessary to change the search path: `dumpbin` is stored in `C:\Program Files\Microsoft Visual Studio\VC98\Bin`. It needs a DLL in `C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin`.
  E.g. with the command "`dumpbin /all /disasm /out:a.x a.obj`" the information in `a.obj` is written to the file `a.x`.
  One can apply `dumpbin` in the same way to executable files.

- Under UNIX, the program `nm` shows the symbol table of an object file.

  Also the GNU program `objdump` might be available to show the contents of the file.

Object Files (5)

- In the object file `a.obj`, the code for `main` looks basically as follows (continued on next slide):

```
0000 push ebp
0001 mv   ebp, esp
0003 sub  esp, 4
0006 push 1
0008 call 00 /* Call of p */
0013 add  esp, 4
0016 mv   dword ptr [ebp-4],eax
0019 mv   eax, dword ptr [ebp-4]
0022 push eax
0023 push 00 /* "The result is: %d\n" */
```


Object Files (6)

- Code for `main` in `a.obj`, continued:

```
0028 call 00 /* Call of printf */
0033 add  esp, 8
0036 xor  eax, eax
0038 mv   esp, ebp
0040 pop  ebp
0041 ret
```

- In addition, the object file contains the following information (continued on next slide):
  - ◇ The symbol `_main` is defined as address 0000 in this text section.

Object Files (7)

- Additional information in the object file, continued:
  - ◇ At address 0009 in this text section the value (address) of the symbol `_p` must be stored.

    The byte at address 0008 contains the `call` instruction. The following four bytes contain the address of the procedure that is to be called.
  - ◇ At address 0024 the value of the symbol `??_C@_0..` must be stored.

    This symbol is defined in the same object file, in the `rdata` section. It is the string constant `"The result is: %d\n"`.
  - ◇ At address 0029, the value of `_printf` must be stored.

Object Files (8)

- In the object file `b.obj`, the code for `p` looks basically as follows:

```
0000 push ebp
0001 mv   ebp, esp
0003 mv   eax, dword ptr [ebp+8]
0006 shl  eax, 1
0008 mv   esp, ebp
0010 pop  ebp
0011 ret
```

- It specifies that the symbol `_p` is defined as address 0000 in this text section.

Code Relocation (1)

- The example shows that in order to produce a single executable program, the linker must move the machine code for e.g. the procedure `_p` from start address `0000` to e.g. `0042` (just after `_main`).

  Actually, the VC++ linker moves all code after the address `400000h` which is marked in the executable file as the image base address. The executable program is then probably loaded at this address into main memory, but it can again be relocated when it is loaded (modern computer hardware usually does not require this).

- An alignment for program code might be required or useful, so that `_p` will probably start at address `0048` (16 byte alignment).

Code Relocation (2)

- "Code Relocation" is to move machine code from one memory address to another one.
- Of course, the linker then adjusts also the value of the symbol `_p` to `0041`.
- The linker must also modify other addresses in the program, e.g. for jumps generated by conditional or loop statements.

  Some compilers avoid this by using PC-relative jumps (PC is the program counter, i.e. the address of the current instruction).

Libraries (1)

- A library is basically only a collection of object files that are archived in one file.

  The standard C library `libc.lib` and its debugging version `libcd.lib` are stored in `C:\Program Files\Microsoft Visual Studio\VC98\Lib`. The command "`lib libcd.lib /list >libcd.x`" writes a list of all object files in `libcd.lib` to the file `libcd.x` (the library contains 638 files). Under UNIX, the program `ar` performs this task. E.g. in order to see the components of the standard library, use `ar t /usr/lib/libc.a`.

- One can extract the object files from a library.

  One can extract an object file from a library with the following command: "`lib` ⟨Library⟩ `/EXTRACT:`⟨MemberName⟩ `/OUT:`⟨File⟩". The member file name must be exactly as specified in the library (including the path). Under UNIX, use "`ar x` ⟨Library⟩ ⟨MemberName⟩".

Libraries (2)

- When the linker processes a library it extracts only the needed object files and links them with the explicitly specified object files.

  When you specify an object file as input to the linker, it always becomes part of the executable program. When you specify a library, the linker selects only the needed object files in the library.

- Needed means that at the point when the linker processes the library, there is a referenced symbol (function/variable) that is defined in this object file.

  For this reason, libraries are usually mentioned on the command line after the object files. Also the exact sequence of libraries may be important if one library calls functions from the other library.

Libraries (3)

- In a library, each object file usually defines only a single function (or one library function plus the auxillary functions that it calls).

  This explains why there are so many different object files in the standard C library.

- The reason for this is that the object files and not the functions are the unit for linking.

  If the entire library were a single object file, it would always be linked completely to the program. By splitting the library into many object files, only the really required machine code is linked to the program.

Libraries (4)

- Of course, library functions often call other (lower level) library functions. Therefore, not only the object files for the functions that the user directly called are linked to the executable program, but also a lot of indirectly called ones.

  In VC++, one can add the option `/verbose` to the field "`Project Options`" under `Project`→`Settings`→`Link`. Then the linker gives detailed information about which object files from which libraries are added to the executable, and which object file referenced a function in this file.

Libraries (5)

- A library often contains an index (symbol table) that tells the linker with object file defines which symbols.

  In this way, the linker does not have to read the entire library in order to find the needed object files. Another advantage is that the object files do not have to be in a specific order. Without the index, linkers often require that functions in a library object file can only call functions in object files that appear later in the library. Under UNIX, the program `ranlib` creates the index.

- Today, programs are often dynamically linked with libraries. Then the actually linking is done at runtime (see below).

Example: Using a Library (1)

- The compiler will tell the linker automatically that it should link with the standard library.
- But suppose we want to compute $\sqrt{2}$:

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

int main()
{   double r;
    r = sqrt(2.0);
    printf("The result is: %lf\n", r);
    return 0;
}
```

Example: Using a Library (2)

- In C, the mathematical functions like `sqrt` are not part of the standard library.

  Actually, in MS VC++, they are.

- The "mathematics" library must be explicitly specified, or one gets an "undefined symbol" error.

  Under UNIX, one adds the option "`-lm`", e.g. "`gcc a.c -lm`". In general, the option `-l`x searches for a library called `lib`x`.a` (or `lib`x`.so.*` for dynamic linking). Alternatively, one can explicitly mention the library: "`gcc a.c /usr/lib/libm.a`" (but using `-lm` is much more common). If the library is not contained in one of the standard directories, and one wants to use `-l`, one can extend the search path for libraries with the option `-L`, e.g. "`gcc a.c -L/home/brass/lib -lxyz`". It will then find `/home/brass/lib/libxyz.a`.

Overview

Make (1)

- Under UNIX, the program **`make`** is used to compile only those source files that need recompilation.

  MS VC++ comes with a program "`nmake`" that seems to have basically the same functionality. But most programmers probably use the graphical development environment instead.

- **`make`** needs a file with dependency rules. This file is normally called **`Makefile`** or **`makefile`**.

  If both files exist, `makefile` is used. If the file has a non-standard name, one must specify it, e.g. "`make -f myprog.mk`".

- **`make`** is not bound to C or program development.

  It can always be used when files can be generated from other files.

Make (2)

- The **`makefile`** consists of a list of dependency rules and macro definitions. Each rule consists of
  - ◇ a goal (or target) A
  - ◇ the names of files $B_1 \dots B_n$ on which A depends (the "dependents"),
  - ◇ and a list of commands $C_1 \dots C_k$:

$$\begin{array}{ll} A: & B_1 \;\; B_2 \; \dots \; B_n \\ & C_1 \\ & \vdots \\ & C_k \end{array}$$

Make (3)

- When make tries to construct the goal A, it will first recursively try to make B_1 to B_n.
- Then it will check whether the file A exists. If it does not, it will execute the commands C_1 to C_k.
- If the file A exists, it will compare the date/time of last modification with the timestamp of B_1 to B_n.
- If one of the files B_i was modified after A, it will execute the commands C_1 to C_k.

  Otherwise (the file A exists and was created/modified after all B_i), make will not execute the commands.

Make (4)

Example Makefile:

- Suppose that the program `myprog` consists of two modules: `main.c` and `stack.c`. Both include `stack.h`.

```
# This is a comment.
myprog: main.o stack.o
        gcc main.o stack.o -o myprog

main.o: main.c stack.h
        gcc -c main.c

stack.o: stack.c stack.h
        gcc -c stack.c
```


Make (5)

Syntax details:

- The goal/target must start in the first column. No spaces are permitted in front of it.
- The list of dependents extends to the end of the line. If one wants to continue it on the next line, one needs to terminate the first line with a backslash "\".
- The commands must be indented by tabulator characters. Alternatively, one can put a semicolon in front of them.

Make (6)

- If one calls "`make`" without arguments, it tries to make the first goal in the file.

  Therefore the first (default) goal should be the program to be constructed. If one wants to build another goal, one can specify it as a parameter, e.g. "`make main.o`".
- Goals do not need to correspond to files.

  If there is no file for a goal, `make` will always execute the associated commands when it tries to make the goal.
- E.g. one can add to the makefile (at the end):

```
clean:
        rm -f main.o stack.o myprog
```

Make (7)

- It is possible to define macros, e.g.

  `OBJECTS = main.o stack.o`

- If one writes later "`$(OBJECTS)`", it is replaced by "`main.o stack.o`"

  The parentheses are necessary unless the macro name is a single letter.

- Some macros might be predefined.

  Especially, under UNIX the environment variables are predefined as macros.

Make (8)

- It is possible to define general rules based on the file extension (suffix). Some such rules are predefined.
- E.g. `make` knows that x`.o` can be constructed from x`.c` by calling the C-compiler:

  `$(CC) -c $(CFLAGS)` x`.c`

- However, dependencies from include files still must be specified.

  For small projects, this can be done manually. For large projects, the list of dependencies is generated by the compiler, e.g. with the command "`gcc -M main.c stack.c`". One can write a shellskript that updates the makefile.

Make (9)

Example Makefile (Improved):

```
PROG   = myprog
OBJECTS= main.o stack.o
CC     = gcc
CFLAGS = -Wall -DDEBUG

$(PROG): $(OBJECTS)
        gcc $(OBJECTS) -o $(PROG)

main.o: stack.h
stack.o: stack.h

clean:
        rm -f $(OBJECTS) $(PROG)
```


Overview

Motivation (1)

- Linking as explained above (static linking) has a number of disadvantages:
  - ◇ Wasted Disk Space: If there are many executable files linked with the same library (and using more or less the same functions), each contains a copy of that part of the library.
  - ◇ Wasted Memory: If several programs containing the same library functions execute at the same time, each needs memory for its own copy of the library function.

Motivation (2)

- Problems of static linking, continued:
  - ◇ Less Flexibility: If a new version of the library is published, applications that were linked with the old version must be explicitly relinked.

    So the user has to get an update of each application instead of installing only an update of the library.
  - ◇ More Difficult Configuration: If there are several versions of a program that differ only in a few modules, the vendor nevertheless must distribute the complete executable for each version.

Motivation (3)

- Problems of static linking, continued:
  - ◇ No user configuration: The user can choose only one of the versions distributed by the software vendor instead of combining the features in which he/she is interested.
  - ◇ No user extension: The user cannot extend the program with functions he/she has developed.
- To be fair, the vendor could distribute the object files and call the linker in the installation routine.

  However, this may reveal too much confidential information.

Dynamic Linking (1)

- In dynamic linking, the library is not part of the distributed executable program.

  So the executable program is not complete (not really executable).
- The compiler will translate a call to a library procedure into an indirect jump via an “import table”.
- The executable program contains
  - ◇ the names of the needed libraries and
  - ◇ names/numbers of the called functions in these libraries together with the corresponding index in the import table.

Dynamic Linking (2)

- When the program starts, the operating system
  - ◇ searches the libraries,
  - ◇ loads them into memory (or gives the program access to them if they are already loaded), and

    When several programs share the same library, a problem is that it must either get the same address or use some form of relative addressing (so that it works at different addresses).
  - ◇ fills the import table with the addresses of the called functions.

Dynamic Linking (3)

- In this way, using a dynamic link library (DLL) is not much different from using a classical (static) library:
  - ◇ When the program is linked, the linker will check that the called functions exist in the library,

    So any undefined functions are still found at link time.
  - ◇ but it will not add the machine instructions for them to the program.
  - ◇ This only happens when the program is executed.

Dynamic Linking (4)

- Dynamic linking has also several drawbacks:
  - ◇ When the program is started, the operation system must be able to find the dynamic link library.

    Windows searches: (1) the directory where the executable program is stored, (2) the current directory, (3) the windows system directory, (`C:\WINDOWS\SYSTEM`), (4) the windows directory (`C:\WINDOWS`), (5) the directories listed in the environment variable "`PATH`". UNIX uses the environment variable "`LD_LIBRARY_PATH`".
  - ◇ There may be version conflicts.

    DLLs are updated, but a program that worked with an older version of the DLL might crash when linked with a newer version. The DLL vendor tries to avoid incompatible updates, but the program may have used undocumented features that are no longer supported.

Dynamic Linking (5)

- There is also another kind of dynamic linking (called explicit linking in Windows) in which
  - ◇ the program asks the operating system at runtime to load a library (`LoadLibrary`),
  - ◇ then queries the address of the procedure to be called (`GetProcAddress`),
  - ◇ and then calls the procedure via this pointer.
- In this way:
  - ◇ The program can react to errors.
  - ◇ The libraries to load may depend on user input.

Dynamic Linking (6)

- In MS VC++, the program `dumpbin` can be used to list dependencies from DLLs.

  "`dumpbin /dependents prog.exe`" lists the DLLs that are required by `prog.exe`. "`dumpbin /imports prog.exe`" lists the single functions (together with their identifying numbers). For DLLs, there is the corresponding option "`/exports`".

- In a sense, DLLs can be seen as extensions of the operating system.

  The operating system also contains a lot of procedures (OS calls) that programs can use, and that are not part of the program.