CHAPTER XVIII

Writing and Compiling Your Programs

This chapter presents techniques to use when writing and compiling programs. You will learn several techniques used by professional C programmers in their everyday programs. In this chapter, you will learn that separating your source code into several files can be helpful in small and large projects alike, especially when you are creating function libraries. You will learn which memory models are available and which you will need to use for the different projects you work on. If you have several source files that make up your application, you will benefit from learning about a utility called MAKE that can help manage your project. You will learn what the difference between a .COM file and an .EXE file is, and a possible advantage to using .COM files.

Additionally, you will learn techniques to use to overcome a typical DOS problem: not enough memory to run your program. Usage of expanded memory, extended memory, disk swapping, overlay managers, and DOS extenders is discussed in an attempt to provide you with several options to remedy the "RAM cram" problem—choose the method that's best for you.

XVIII.1: Should my program be written in one source file or several source files?

Answer:

If your program is extremely small and focused, it is perfectly OK to contain all the source code within one .c file. If, however, you find yourself creating a lot of functions (especially general-purpose functions), you will want to split your program into separate source files (also known as modules).

The process of splitting your source code into several source files is known as *modular programming.* Modular programming techniques advocate the use of several different focused modules working together to make up a complete program. For instance, if your program has several utility functions, screen functions, and database functions, you might want to separate the functions into three source files that make up the utility module, screen module, and database module.

By putting your functions in separate files, you can easily reuse your general-purpose functions in other programs. If you have several functions that can be used by other programmers, you might want to create a function library that can be shared with others (see FAQ XVIII.9).

You can never have "too many" modules—you can create as many for your program as you see fit. A good rule of thumb is to keep your modules focused. Include only functions that are logically related to the same subject in the same source file. If you find yourself writing several nonrelated functions and putting them in the same file, you might want to pause to look at your program's source code structure and try to create a logical breakdown of modules. For example, if you are creating a contact management database, you might want to have a structure like this:

Module Name	*Contains*
Main.c	The `main()` function
Screen.c	Screen management functions
Menus.c	Menu management functions
Database.c	Database management functions
Utility.c	General-purpose utility functions
Contact.c	Functions for handling contacts
Import.c	Record import functions
Export.c	Record export functions
Help.c	On-line help support functions

Cross Reference:

XVIII.10: My program has several files in it. How do I keep them all straight?

XVIII.2: What are the differences between the memory models?

Answer:

DOS uses a segmented architecture to address your computer's memory. For each physical memory location, it has an associated address that can be accessed using a segment-offset method. To support this segmented architecture, most C compilers enable you to create your programs using any of the six memory models listed in the following table:

Memory Model	*Limits*	*Pointer Usage*
Tiny	Code, data, and stack—64KB	Near
Small	Code—64KB	Near
	Data and stack—64KB	Near
Medium	Code—1 megabyte	Far
	Data and stack—64KB	Near
Compact	Code—64KB	Near
	Data and stack—1 megabyte	Far
Large	Code—1 megabyte	Far
	Data and stack—1 megabyte	Far
Huge*	Code—1 megabyte	Far
	Data and stack—1 megabyte	Far

* Note that in the Huge memory model, static data (such as an array) can be larger than 64KB. This is not true in all the rest of the memory models.

The Tiny memory model is extremely limited (all code, data, and stack must fit in 64KB); it is most often used for the creation of .COM files. The Huge memory model imposes a significant performance penalty because of the way it has to "fix up" memory addresses; it is rarely used.

Cross Reference:

XVIII.3: What are the most commonly used memory models?

XVIII.4: Which memory model should be used?

XVIII.3: What are the most commonly used memory models?

Answer:

The most common are the Small, Medium, and Large memory models. The Tiny memory model is typically used only for creation of .COM files, which is somewhat rare in today's world of high-powered machines. The Compact memory model allows your program to have very little code and a lot of data. This, too, is uncommon in today's business place, because very often you will find significant amounts of code where there are significant amounts of data. The Huge memory model is somewhat inefficient because of the memory addressing scheme it imposes, and it is also a rarity.

Typically, you should use the Small, Medium, or Large memory models, depending on the size of your program. For a small utility program, the Small memory model might be suitable. This memory model enables you to have 64KB of code and 64KB for your data and stack. If your program has slightly larger data requirements than this, you might want to use the Medium memory model, which enables you to have up to 1 megabyte of addressable data space. For larger programs, you will want to use the Large memory model, which enables you to have 1 megabyte of code and 1 megabyte of data and stack space.

If you are writing a Windows program or using a 32-bit compiler, you will use the Small memory model. This is because such environments are not restricted to the segmented architecture of DOS programs.

Cross Reference:

XVIII.2: What are the differences between the memory models?

XVIII.4: Which memory model should be used?

XVIII.4: Which memory model should be used?

Answer:

If you are going to create a .COM file, the Tiny memory model must be used. All code, data, and stack space must fit in 64KB. This memory model is popular among small utility programs. The Small memory model is also used for relatively small programs, except that you are not limited to a total of 64KB for your entire program. In the Small memory model, you can have 64KB for your code space and 64KB for data and stack usage. Besides being used for small programs, the Small memory model is also used in environments such as Windows and for 32-bit compilers because memory addressing is not limited to DOS's 16-bit constraints.

If your program has a relatively large amount of code but relatively small amounts of static data, you can choose to write your program with the Medium memory model. If your program is extremely large (requiring many modules, code, and data), you might want to use the Large memory model. This memory model is most often used for writing business applications in DOS.

Use of the Compact and Huge memory models is much less common than use of the Small, Medium, and Large memory models. The Compact memory model enables you to have a large amount of static data but a relatively small (64KB or less) amount of code. Programs that fit this model are rare and are typically restricted to conversion programs that have large amounts of static translation tables that must be stored in memory. The Huge memory model is identical to the large memory model, except that the Huge memory model allows static data to be larger than 64KB. Like the Compact memory model, the Huge memory model is rare, primarily because its usage imposes a significant performance hit. Because of its relatively inefficient performance, you should avoid using the Huge memory model unless you absolutely must have an array or some other static data that is larger than 64KB. Keep in mind that arrays and other programming constructs can be allocated dynamically at runtime by using functions such as `malloc()` and `calloc()`, and they do not necessarily have to be static in nature.

Cross Reference:

XVIII.2: What are the differences between the memory models?

XVIII.3: What are the most commonly used memory models?

XVIII.5: How do you create a .COM file?

Answer:

Creating a .COM file is accomplished by compiling your program with the Tiny memory model and using special linker commands to make the output extension .COM rather than the normal .EXE extension. Keep in mind that for your program to qualify for a .COM file, all code, data, and stack must be able to fit in 64KB. This memory model is typically restricted to only the smallest of programs, usually programs such as TSRs and small utility programs.

Each compiler has a different method of creating .COM files. You should refer to your compiler's documentation for information regarding which compiler or linker switches you need to use to create a .COM file rather than an .EXE file.

Cross Reference:

XVIII.6: What is the benefit of a .COM file over an .EXE file?

XVIII.6: What is the benefit of a .COM file over an .EXE file?

Answer:

A .COM file is limited to 64KB for all code, data, and stack storage and therefore is limited to small applications such as utility programs and TSRs (terminate-and-stay-resident programs). One distinct advantage of a .COM file over an .EXE file is that .COM files load faster than .EXE files.

A .COM file is also known as a "memory image" file because it is loaded directly into memory with no required "fixups." An .EXE file contains special fix-up instructions inserted by the linker into the file's header. These instructions include a relocation table used to manage the different parts of the executable program. A .COM file does not contain any of these instructions or a relocation table, because the entire program can fit into 64KB. Thus, DOS does not need to parse through any fix-up code, and the .COM file loads faster than an .EXE file.

.COM files are usually simplistic and are somewhat limited in what they can accomplish. For instance, you cannot allocate memory from the far heap from a .COM file.

Cross Reference:

XVIII.5: How do you create a .COM file?

XVIII.7: Are all the functions in a library added to an .EXE file when the library is linked to the objects?

Answer:

No. When the linker is invoked, it will look for "unresolved externals." This means that it will poll your library files for functions that were not defined in your source code files. After it finds an unresolved external function, it pulls in the object code (.obj) which contains that function's definition. Unfortunately, if this function was compiled with a source file that contained other function definitions, those functions are included also. You therefore might have unwanted and unneeded code unnecessarily pulled into your executable information. This is why it is important to keep your library functions contained within their own source file—otherwise, you might be wasting precious program space. Some compilers contain special "smart" linkers that can detect unneeded functions such as these and discard them so that they don't make their way into your program.

Here is an example: Suppose that you have two source files, libfunc1.c and libfunc2.c. Each contains functions you want to put in a library.

The source file libfunc1.c contains the following two functions:

```
void func_one()
{
     ...
}

void func_two()
{
     ...
}
```

The source file libfunc2.c contains the following function:

```
void func_three()
{
     ...
}
```

Now suppose that you have compiled these two source code files into a library named myfuncs.lib. Suppose that a program linked with myfuncs.lib contains a call to `func_one()`. The linker will search the myfuncs library to pull in the object code that contains the definition of the `func_one()` function. Unfortunately, the `func_one()` function was compiled with the same source file that contains the definition for the `func_two()` function, and the linker will be forced to pull in the `func_two()` function even though your program doesn't use it. Of course, this assumes that `func_one()` does not contain a call to `func_two()`. If a program were to contain a call to `func_three()`, only the object code for `func_three()` would be pulled in because it was compiled in its own source file.

Generally, you should keep library functions contained within their own source files. This organization helps your programs to be more efficient because they will be linked only with the functions they really need, and not other functions they don't need. This also helps in a team development situation in which source code files are continually checked in and checked out. If a programmer is going to perform maintenance on a

function that is contained within its own source file, they can focus on that one function. If the source file were to contain several other function definitions that needed maintenance, other programmers would not be able to check out the other functions because they are contained in one source file.

Cross Reference:

XVIII.8: Can multiple library functions be included in the same source file?

XVIII.9: Why should I create a library?

XVIII.8: Can multiple library functions be included in the same source file?

Answer:

You can define as many functions as you want in the same source file and still include them in a library—however, this technique has serious disadvantages when it comes to linking your programs and sharing source files in a team development environment.

When you include more than one library function in the same source file, the functions are compiled into the same object (.obj) file. When the linker links one of these functions into your program, all the functions in the object file are pulled in—whether or not they are used in your program. If these functions are unrelated (do not have calls to each other within their definitions), you will be wasting precious program space by pulling unneeded code. See FAQ XVIII.7 for an example. This is one reason why it is better to put library functions in their own separate source files.

Another good reason to put library functions in their own source files is for code sharing in a team development environment. Using separate source files enables programmers to check out and check in individual functions, instead of locking others out of being able to make changes to several functions contained in one source file.

Cross Reference:

XVIII.7: Are all the functions in a library added to an .EXE file when the library is linked to the objects?

XVIII.9: Why should I create a library?

XVIII.9: Why should I create a library?

Answer:

Creating a function library enables you to put reusable functions in a place where they can be shared with other programmers and programs. For instance, you might have several general-purpose utility functions

that are used in several of your programs. Instead of duplicating the source code for all of these different programs, you can put these functions in a centralized function library and then link them into your program when the linker is invoked. This method is better for program maintenance, because you can maintain your functions in one centralized place rather than several places.

If you are working in a team environment, putting your reusable functions in a library allows other programmers to link your functions into their programs, saving them from having to duplicate your effort and write similar functions from scratch. Additionally, in large projects that involve several modules, a function library can be used to contain "framework" support functions that are used throughout the application.

Your compiler includes a library manager (typically named LIB.EXE or something similar) that can be used to add and delete object code modules (.obj's) from function libraries. Some compilers enable you to maintain your libraries from within their integrated development environments without having to invoke a library manager manually. In any case, you should refer to the answers to FAQ XVIII.7 and XVIII.8 for important information regarding the creation of libraries and good techniques to adhere to.

Cross Reference:

XVIII.7: Are all the functions in a library added to an .EXE file when the library is linked to the objects?

XVIII.8: Can multiple library functions be included in the same source file?

XVIII.10: My program has several files in it. How do I keep them all straight?

Answer:

Your compiler includes a MAKE utility (typically called MAKE.EXE, NMAKE.EXE, or something similar) that is used to keep track of projects and the dependencies of source files that make up those projects. Here is an example of a typical MAKE file:

```
myapp.obj:      myapp.c             myapp.h
                cl -c myapp.c

utility.obj:    utility.c           myapp.h
                cl -c utility.c

myapp.exe:      myapp.obj           utility.obj
                cl myapp.obj        utility.obj
```

This example shows that myapp.obj is dependent on myapp.c and myapp.h. Similarly, utility.obj is dependent on utility.c and myapp.h, and myapp.exe is dependent on myapp.obj and utility.obj. Below each dependency line, the compiler command to recompile or relink the dependent object is included. For instance, myapp.obj is re-created by invoking the following command line:

```
cl -c myapp.c
```

In the preceding example, myapp.obj is recompiled only if myapp.c or myapp.h has a time stamp later than myapp.obj's time stamp. Similarly, utility.obj is recompiled only when utility.c or myapp.h has a time stamp later than utility.obj's time stamp. The myapp.exe program is relinked only when myapp.obj or utility.obj has a later time stamp than myapp.exe.

MAKE files are extremely handy for handling large projects with many source file dependencies. MAKE utilities and their associated commands and implementations vary from compiler to compiler—see your compiler's documentation for instructions on how to use your MAKE utility.

Most of today's compilers come with an integrated development environment, in which you can use project files to keep track of several source files in your application. Having an integrated development environment frees you from having to know the intricacies of a MAKE utility and enables you to easily manage the source files in your project. The integrated development environment automatically keeps track of all dependencies for you.

Cross Reference:

XVIII.1: Should my program be written in one source file or several source files?

XVIII.11: I get the message *DGROUP: group exceeds 64K* during my link. What's wrong?

Answer:

If you see this error message while linking your program, the linker is indicating that you have more than 64KB of near data (static data elements, global variables, and so on) in your data (DGROUP) segment. You can remedy this situation in a few ways:

- Eliminate some of your global variables.
- Decrease your program's stack size.
- Use dynamic memory allocation techniques to dynamically allocate data elements instead of defining them as static or global.
- Declare data elements specifically as far rather than near.

Eliminating some of your global variables will probably require some rework on your part as to the inherent design of your program, but it will be worth it when all is said and done. Global variables by nature tend to be somewhat of a maintenance nightmare and should be used only when absolutely necessary. If you have allocated a lot of space to be used as stack space, you might want to experiment with lowering the stack space size to see whether you can gain memory that way. If you are using a lot of static data in your program, try to think of a way you could possibly rework your static data and allocate it dynamically rather than statically. This technique will free up the near heap and enable you to allocate data from the far heap instead (see FAQ XVIII.15 for a discussion on near and far heap space).

Cross Reference:

XVIII.12: How can I keep my program from running out of memory?

XVIII.12: How can I keep my program from running out of memory?

Answer:

If you are using a lot of static data, you might want to think about using dynamic memory allocation instead. By using dynamic memory allocation (with the `malloc()` and `calloc()` functions), you can dynamically allocate memory when you need it and release it (via the `free()` function) when it is no longer needed. This helps in a couple of ways. First, dynamic memory allocation allows your program to be more efficient because your program uses memory only when necessary and uses only the memory it really needs. You don't have a lot of unused memory unnecessarily being taken up by static and global variables. Second, you can check the return value of the `malloc()` and `calloc()` functions to trap for situations in which you might not have enough memory.

If your program is extremely large, you might want to use an overlay manager or a DOS extender, or you might want to use alternative memory allocation schemes such as EMS or XMS (see FAQs XVIII.13 and XVIII.14 for further discussion on these topics).

Cross Reference:

XVIII.13: My program is too big to run under DOS. How can I make it fit?

Answer:

When your application has grown too large for DOS (over 640KB), there are two good ways to give your program more memory. One way is to use an *overlay manager*. An overlay manager will manage the modules (.obj files) of your program and read them in from disk and discard them as needed. This way, your program can be several megabytes in size and still fit in a computer that has only 640KB of memory available. Some advanced overlay managers enable you to determine module "groups" that you would like to be read in and discarded all together. This helps you fine-tune your application for performance reasons. Other less advanced overlay managers do not have this feature and do not enable you to fine-tune which overlaid modules should be treated as a group.

Another way to get more memory for your application is to use a *DOS extender*. A DOS extender is a special application that uses the protected mode features of 386, 486, and newer computers to access several megabytes of memory in one flat address space. When your program is linked with a DOS extender, the DOS extender code becomes a part of your program's start-up code. When your program is invoked, the DOS extender is loaded and your program falls under the control of the DOS extender. All memory allocation calls are routed through the DOS extender, thereby enabling you to bypass DOS and let the extender handle the intricacies of allocating memory above the 640KB threshold.

Unfortunately, DOS extenders have some definite disadvantages. One disadvantage is that most DOS extenders have runtime royalty fees that apply when you distribute your programs. This can be quite costly, especially if you have many users. A few compilers come with royalty-free DOS extenders, but this feature is typically the exception rather than the norm. Another disadvantage of using a DOS extender is that its operation typically requires you to change your source code to access the extender's application program interface (API) instead of using DOS calls.

Overlay managers do not typically require runtime fees, so they are more cost efficient and less expensive than DOS extenders. Additionally, you rarely need to change your source code to use an overlay manager. Most of the time, the use of an overlay manager is transparent to the program.

Cross Reference:

XVIII.11: I get the message `DGROUP: group exceeds 64KB` during my link. What's wrong?

XVIII.12: How can I keep my program from running out of memory?

XVIII.14: How can I get more than 640KB of memory available to my DOS program?

XVIII.15: What is the difference between near and far?

XVIII.14: How can I get more than 640KB of memory available to my DOS program?

Answer:

When you find yourself in a memory-crunch situation, needing to use more than 640KB of memory in a DOS program, you can use a few good methods of getting more memory available. One way is to use *disk swapping.* Disk swapping means that you write data elements that are stored in memory to disk when you do not need them. After writing a data element (variable, array, structure, and so forth) to disk, you can free up the memory that was used by that data element (by using the `free()` function) and thus have more memory available to your program. When you need to use the data element that was swapped to disk, you can swap out another data element from memory to disk and read the previously swapped variable back in from disk. Unfortunately, this method requires a lot of coding and can be quite tedious to implement.

Another good way to get more than 640KB of memory available to your DOS program is to use an alternative memory source—EMS (expanded memory) or XMS (extended memory). EMS and XMS, which refer to two ways of allocating memory above the 640KB region, are explained in separate paragraphs in the following text.

EMS stands for Expanded Memory Specification. This is a method developed by Lotus, Intel, and Microsoft for accessing memory above the 1 megabyte region on IBM-compatible machines. Currently, two versions of this specification are used: LIM 3.2 and LIM 4.0. The newer version, LIM 4.0, overcomes some of the limitations of LIM 3.2. Expanded memory is enabled by the installation of an expanded memory manager (such as EMM386.EXE included with DOS). Your program makes calls to the expanded memory manager to request blocks of expanded memory. The expanded memory manager uses a technique called bank switching to move memory temporarily from above the 1 megabyte region to an empty region in the upper memory area between 640KB and 1 megabyte. Bank switching involves taking a memory allocation request from the application program and allocating 16KB of upper memory area at a time to keep track of memory that is addressed above the 1 megabyte region.

Extended memory is enabled by the installation of an extended memory manager (such as HIMEM.SYS included with DOS). Your program makes calls to the extended memory manager to request extended memory blocks (EMBs). No "bank switching" technique is used for requesting extended memory. Your program simply makes a function call to the extended memory manager to request a block of memory above the 1 megabyte region. Unfortunately, code cannot be executed above the 1 megabyte region under DOS; therefore, you cannot execute code stored in extended memory. Similarly, you cannot directly address data stored in extended memory, so many programmers like to set up a "buffer area" in conventional memory (below 640KB) to provide a swap area between conventional and extended memory.

The techniques used for expanded memory are older and somewhat outdated. Expanded memory was popular when DOS-based machines first came out that had add-on expanded memory boards attached. Using expanded memory techniques is somewhat slower than using extended memory techniques. In fact, many of today's PC configurations eliminate expanded memory altogether by including the NOEMS flag in the EMM386.EXE entry of the config.sys file. Most modern programs have abandoned the older expanded memory techniques for the newer extended memory techniques.

If your program needs to address above the 1 megabyte region, you should use extended memory rather than expanded memory. When you use extended memory, your programs will have greater stability and perform faster than if you had chosen to use expanded memory.

The specific steps of implementing extended and expanded memory are beyond the scope of this book. Explaining how to address memory with these techniques would probably require a separate chapter. Instead, you can obtain the EMS (Expanded Memory Specification) and XMS (Extended Memory Specification) documents directly from Microsoft or download them from a network service such as CompuServe. These documents detail the EMS and XMS application programming interface (API) and show you in detail how to use each technique.

Cross Reference:

XVIII.11: I get the message `DGROUP: group exceeds 64KB` during my link. What's wrong?

XVIII.12: How can I keep my program from running out of memory?

XVIII.13: My program is too big to run under DOS. How can I make it fit?

XVIII.15: What is the difference between near and far?

XVIII.15: What is the difference between near and far?

Answer:

DOS uses a segmented architecture to address your computer's memory. For each physical memory location, it has an associated address that can be accessed using a segment-offset method. For instance, here is a typical segmented address:

```
A000:1234
```

The portion on the left side of the colon represents the segment (A000), and the portion on the right side of the colon represents the offset from that segment. Every program under DOS accesses memory in this manner—although the intricacies of addressing with the segment-offset method are often hidden from the casual C programmer.

When your program is executed, it is assigned a default data segment that is put in the data segment (DS) register. This default data segment points to a 64KB area of memory commonly referred to as near data. Within this near data area of memory, you will find your program's stack, static data, and the near heap. The near heap is used for allocating global variables and other data elements you need at program start-up. Any data allocated from this area is called near data. For instance, consider the following program, which allocates 32KB of near data from the near heap at program start-up:

```
/* Note: Program uses the Medium memory model... */

#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <stdlib.h>
#include <dos.h>

void main(void);

void main(void)
{

     char* near_data;

     near_data = (char*) malloc((32 * 1024) * sizeof(char));

     if (near_data == (char*) NULL)
     {
          printf("Whoopsie! Malloc failed!\n");
          exit(1);
     }

     strcpy(near_data,
            "This string is going to be stored in the near heap");

     printf("Address of near_data: %p\n", &near_data);

     free(near_data);

}
```

In the preceding example, `near_data` is a character pointer that is assigned a 32KB block of memory. By default, the 32KB block of memory is allocated from the near heap, and the resulting 16-bit address is stored in the character pointer `near_data`.

Now that you are aware of what near data is, you are probably wondering what far data is. Quite simply, it is any data that resides outside of the default data segment (the first 64KB of data memory). Here is an example program that allocates 32KB from the far data area (or far heap, as it is commonly called):

```
/* Note: Program uses the Medium memory model... */

#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <stdlib.h>
#include <dos.h>

void main(void);

void main(void)
{

     char far* far_data;

     far_data = (char far*) farmalloc((32 * 1024) * sizeof(char));

     if (far_data == (char far*) NULL)
     {
          printf("Whoopsie! Far malloc failed!\n");
          exit(1);
     }

     _fstrcpy(far_data,
              "This string is going to be stored in the far heap");

     printf("Address of far_data: %Fp\n", &far_data);

     farfree(far_data);

}
```

In this example, the far character pointer is assigned a 32-bit address reflecting a 32KB area of free memory in the far heap. Notice that to explicitly allocate from the far heap, a far pointer must be used, and hence the far modifier is added to the character pointer definition. Also note that some of the functions (`farcoreleft()`, `farmalloc()`, `farfree()`) are different for allocating from the far heap as opposed to the near heap.

The far heap usually contains much more free memory than the near heap, because the near heap is limited to 64KB. If you compile and run the previous examples on your computer, you will find that the first example (which allocates from the near heap) has approximately 63KB of memory available. The second example (which allocates from the far heap) has approximately 400KB to 600KB (depending on your computer's configuration) of memory available. Thus, if your program requires a lot of memory for data storage, you should use the far heap rather than the near heap.

Whatever memory model you use (with the exception of the Tiny memory model), you can use the near and far modifiers and their corresponding near and far functions to explicitly allocate memory from the near and far heap. Using near and far data wisely will help your programs run more efficiently and have less risk of running out of memory.

Note that the concept of near and far data is unique to personal computers running DOS because of the segmented architecture scheme used by DOS. Other operating systems such as UNIX or Windows NT use flat memory models, which impose no near or far limitations.

Cross Reference:

XVIII.11: I get the message `DGROUP: group exceeds 64KB` during my link. What's wrong?

XVIII.12: How can I keep my program from running out of memory?

XVIII.13: My program is too big to run under DOS. How can I make it fit?

XVIII.14: How can I get more than 640KB of memory available to my DOS program?