1. INTRODUCTION

The major challenge for present technology is to generate software more efficiently. Computers are being used in essentially all areas of human activities; even products that seemingly have nothing to do with computers contain a significant portion of computer related cost.

More efficient software means three things: coding and debugging programs faster; generating programs with less errors; and producing cleaner, more readable code, which is easier to maintain.

This library provides major improvements in all three of these areas. Current experience indicates that, with this library, algorithmic programs can be developed 2-4 times faster (coding and debugging). The increase in software productivity is even bigger for in-core databases (5-10 times faster development and testing compared to hand coded C++). The resulting code is more readable, and therefore easier to maintain or transfer to another programmer.

When you use this library, software productivity is increased without sacrificing run-time performance. Programs developed with this library are just as fast as those developed with hand coded C++, and have no memory overhead. The class generator carries the overhead for data management and converts the original source into efficient, simple C++ code which does not use offsets or indexing for pointers.

Also note that there is also a C version of this library. We prefer to use C++, but if you or some of your friends need advanced data structures and persistency which handles almost as easily as C++, contact Code Farms.

This library is not just a package for the management of data, it is a new methodology, which extends the idea of structured programming. It requires not only program logic, but also the data to be structured and well organized. This library brings order into your thinking process; both you and the program will benefit.

The library helps in the development of solid production code, but it is also useful for rapid prototyping. Examples with complicated data structures including graphs and hash tables can be coded and tested within hours. For this reason, this library is invaluable for software engineering classes; students can test complicated applications without losing time on implementation details.

This text starts with the most important but simple to understand concepts, and leaves the more advanced features for later. You can read any chapter independently, but if you are a new user, reading the chapters one by one will give you the best introduction to the library.

Most programmers are able to use this library within one day. Within a week, you will be confident enough to use some advanced features.

If you are a new user, read only Chap. 1 to 15, and leave the remaining parts for later. You may never attempt to expand the existing library or to use the program in multi-user mode, and too much

information would clutter your perception in the beginning.

As you read this text, try to code simple test programs as soon as you can. After reading the example in Chap.7, type your own program into the *orgC/test* directory as *myTest*.c. Then *try1 myTest* will run the class generator, and *ctry2 myTest* will compile your code. If you have any problems, see Chap. 4, the section on the test directory. Most likely, you will have to set up a proper link to your compiler.

Once you get through your first program, the spell will be broken, and progress will be rapid.

Don't be frustrated if you get a lot of class generator and compiler errors. When you program with this library, you will have a harder time to get your program through the compiler, but once it compiles, there will be very little or no run-time debugging.<\p>

If you don't plan to use the library right away, and you are only curious about it, start with Chap.6 and Chap.7. They will explain what this library is, and will lead you through a complete example.

Other data management packages come with big binders, which describe a variety of functions available from their libraries. The size of these binders may be impressive, but such systems are difficult to use. Nobody can remember all those functions. This library is just the opposite. It includes all the usual data structures, and it requires only a few pages to describe them (Chap. 11). Generic functions such as *add()* or *del()* are easy to remember and apply to most organizations in the library.

A part of the library installation is the *test* directory, which contains examples of every feature presently in the library. That has two advantages:

- If you are not sure about how to use some organization or function, you can always find an example by using *grep*.
- By executing *cregr* in that directory, you can run through all these examples and verify that everything is functioning properly.

The documentation has three parts:

- The User's Guide - which you are reading, is shipped, in printed form, with the tape or floppy as a part of the software.
- The Reference Guide. You can generate this document yourself, see Chap. 12.
- On-line help. Without a fixed format or menu, you can query about organizations, functions, and macros currently in the library.

The last chapter (Chap.19) contains a record of new features added to library in recent versions.

The library evolved gradually and includes the experience of many people who worked with me over the past 15 years. In particular, I would like to express my gratitude to Mike Yamin(Bell Labs), Ken Keller (Frame Tech.), Mark Bales (CADENCE), Mikael Palzcewski (CADENCE), and the late Rick

Stockburger (BNR). I would also like to thank Hans Gethoffer (ZUKEN, Japan), Carl Seaquist (AT&T Bell Labs), and Bill Williams (BNR) for suggestions that lead to substantial improvements of the program.

Just to avoid confusion, I would like to mention that the graphical CASE tool called "Soukup Method", which was developed in Europe, has nothing to do with Code Farms' libraries. The similarity of authors' names is purely coincidental.

Note that, throughout the documentation, instead of saying "this library", we use the acronym OrgC++ (or OrgC for the C version of the library). This name has historical reasons. In 1989, the library was originally called "Organized C Library". At that time, only the C version was available, and the name emphasized the main purpose - the management of data organizations and, in general, bringing more organization into the software design process. OrgC was the abbreviation for Organized C. Unfortunately, some programmers found the name misleading; some thought it was a new language, while others thought that Organized C and Objective C were the same thing. After some variations of the name, the package is now sold as C/C++ DATA Object Library which, I believe, better explains the purpose of the whole package.

« Title Page

Chapter 2. Installing Floppy or Tape »

2. INSTALLING FLOPPY OR TAPE

2.1 Installing OrgC++ under DOS

Required configuration:

You need an IBM AT compatible machine (386 and up) with a hard drive (5 MB free), and 2MB of RAM. This means that, basically, the configuration which is sufficient for your C or C++ compiler will be sufficient for our library. We assume that you have DOS 3.2 or higher.

Compilers to use:

Microsoft Visual C++ 4.0 or higher
Watcom C/C++ 10.0 or higher
ZORTECH C++ Ver.3.0 or higher
Borland C++ Ver.4.5 or higher

Installation:

The library ships on a single 1.44MB(3.5") diskette. The installation procedure is simple. You insert the diskette and type:

A:
INSTALL

The installer will guide you through the installation procedure which takes just a few minutes.

If you get ERROR 2104, look at the instruction at Chap.2.6.

Compiling the library

After you install the library, you have all the source files in place, but you have to compile them with the compiler you are planning to use. For most compilers, the scripts that recompile individual directories are already part of the library. In each directory, the README file tells you which script file to use.

In order to run with the library, you have to compile orgc and orgc\lib directories. Do not worry about directories orgc\macro, orgc\docum, and orgc\test, you may compile them later when considering some advanced uses of the library.

For example, in order to compile with Borland C++ 4.52 using the large memory model, you do this:

cd c:\orgc
make
cd lib
b4pmakel

In order to compile with Microsoft Visual C++ 4.0 or higher, enter the MS DOS window, and type this:

cd c:\orgc
msft
cd lib
msft ... use msftd when debugging

In order to use VC++ from the compile line (e.g. from the DOS window), the PATH and environment variables must be set properly. Without this, you will not be able to run the scripts that compile orgC++ and run the regression test.

Most versions of Windows (and of the VC++ compilers) allow you to insert this information into the c:\autoexec.bat file. For example:

```
set MSDevDir=C:\PROGRA~1\MICROS~3\COMMON\MSDev98
set INCLUDE=C:\PROGRA~1\MICROS~3\VC98\INCLUDE
set LIB=C:\PROGRA~1\MICROS~3\VC98\LIB
set PATH=%PATH%;C:\PROGRA~1\MICROS~3\VC98\Bin;
C:\PROGRA~1\MICROS~3\COMMON\MSDev98\Bin
```

Under some versions of Windows you can also set these variable by going to Control Pannel, System, Advanced, Environment Variables. Under WinXP this is the only way to do this:

```
include = C:\Program Files\Microsoft Windows Studio\VC98\include
lib = C:\Program Files\Microsoft Windows Studio\VC98\lib
MSDevDir = C:\Program Files\Microsoft Windows Studio\Common\MSDev98
path = C:\Program Files\Microsoft Windows Studio\VC98\bin
```

For more details on how to run under Microsoft Visual C++, see Chap.2.5.

Note that all script files automatically use appropriate environment files (see Chap.5); also the important file orgc/zzcomb.h is automatically generated during the process.

If the compiler which you are planning to use is not listed in the README files, contact Code Farms. We will help you to customize one of the existing script files and to create your own environment file.

After you install the software, test it as shown in the second half of Chap.5.

2.2 Installing OrgC++ under UNIX

OrgC++ runs on a variety of hardware. On a SUN3 or SUN4 you load the streamer tape in the following way:

```
mkdir orgC
cd orgC
tar -xvf /dev/rst8
```

On other machines, only the device is different:

SUN3	rst8
SUN4	rst8
HP9000	rct
Masscomp	rctp
Sony News	rtu00

Recompiling manually

When you un-tar the tape, the entire library is already compiled for SUN OS4.1 (SPARC), regular SUN C and the Sun C++ Ver.2.0. If you run with a different compiler (or a newer version of the C++ compiler), you have to recompile the library. Script files are provided for the most frequently occuring combinations.

When using the SUN C and C++ compilers, replace the line *#define SUN* in files *orgc/lib/env*s.h* as follows. It does not matter whether you run under SUN OS or under Solaris:

```
#define SUN ... for C++ Ver.2.0
#define SUN2_1 ... for C++ Ver.2.1
#define SUN3_0 ... for C++ Ver.3.0 or higher
```

Then type this:

```
cd orgc
jsmake
```

cd lib
jscmake

When using the GNU compiler, type this:

cd orgc
jsgmake
cd lib
jsgmake

When running with Silicon Graphics computer, type this:

cd orgc
jssgmake
cd lib
jssgmake

These runs also re-generate important file orgc/zzcomb.h. Script files for HP, AIX, and other UNIX environments are listed in the README files. If you cannot find your platform/compiler combination in the README file, you will have to set up your special script files and the environ.h file. Call Code Farms, and we will help you to modify one of the existing files.

When you begin to use the library, there is no need to recompile in directories *orgc/docum* and *orgc/macro*. If, later on, you need programs stored in these directories, invoke the script files as indicated in the README files. For example, in order to recompile these directories with the SUN compiler, type:

cd orgc/macro
jsmake
cd orgc/docum
jsmake

Today, relatively few workstations are equipped with a tape drive. Many SUN users load new software from 3.5" diskettes or using ftp over the network. If you opted for 3.5" diskettes instead of the tape, you received several diskettes that contain the full source, which is not compacted, and is organized in several subdirectories. Copy the entire directory tree onto your hard disk.

After you install the software, test it as shown in the second half of Chap.5.

LINUX and GNU

Download the tar file via ftp, or install the DOS diskette (make sure the files are treated as UNIX and not DOS files). The environment and compilation is same as for SUN/GNU combination, but the ready to use batch files are slightly different for LINUX:

lgmake
cd lib
lgmake
cd ../test
lgregr

2.3 Installing OrgC++ on Macintosh

The OrgC++ library is distributed on DOS diskettes, and you need either a Macintosh that can read DOS diskettes, or you must transfer the library over a network. In either case, you must recompile the library. The following sequence assumes that you installed the library under the root directory called *orgc*:

(1) directory *:orgc*

immake
zzcomb

(2) directory *:lib*

jmmake .....(for the C version of the library), or
jmcmake .....(for the C++ version of the library).

If you get to this point without errors, you are ready to run. The remaining steps are only for the auxiliary functions (documentation and library updates):

(3) directory *:macro*

jmmake

(4) directory *:docum*

jmmake

After you install the software, test it as shown in the second half of Chap.5.

2.4 Microsoft Integrated Environment

Enter the MS-DOS Command Prompt , and install, compile, and test the library using the appropriate script files as shown in Chap. 2.1. There is no need (and it is actually more complicated) to do this in the visual environment.

Some programmers use the Windows enviroment for better memory management. You should be aware

that all allocation in the Code Farms library is channelled through a single call to *malloc()* or *calloc()*, see file *lib\msgs.c*. The *new()* operator is overloaded by a call to this function. The purpose of this arrangement is to provide entry for private allocation schemes.

If you don't make any special arrangements for memory allocation, you may be limited by the Windows memory management scheme. Windows puts an upper limit on the number of "handles" that can be allocated (approximately 8k). Depending on how your compiler implements *malloc()* etc., you may not be able to create more than 8k objects!!

There are three ways to overcome this limitation:

(1) Change the calls to *malloc()*, *calloc()*, *realloc()*, and *free()* in file *lib\msgs.c* to the Windows allocation functions (*GlobalAlloc()* etc.).

(2) Code your own version of *malloc()*, *calloc()*, *realloc()*, and *free()*, using Windows allocation functions (*GlobalAlloc()*). Then link these functions before Borland's or other library.

(3) Use a commercial memory allocator like SmartHeap from MicroQuill Corp (tel 206-535-8218), and link it again before Borland's library. The package provides replacement for *malloc()*, *calloc()*, *realloc()*, *free()*, *new()*, and *delete()* with essentially no limitations. Note that since the OrgC++ library also overloads operators *new()* and *delete()*, it is essential that you link the OrgC++ library first, then the commercial allocator, and then Borland's or other library.

If you use Visual C++ from Microsoft Corp., your entire environment is driven by menu items and graphical displays, and is subject to unusual restrictions not present in other compilers. Visual C++ is not a normal compiler, but rather a windows developement framework. The following items are important when using the Code Farms library with Visual C++:

ITEM 1: Make sure that autoexec.bat includes correct PATH, INCLUDE, and LIB statements:

```
PATH .....;C:\msdev\bin
SET INCLUDE=c:\msdev\include
SET LIB=c:\orgc\lib;c:\msdev\lib
```

ITEM 2: The compilation is the same, whether you run under Win3.1, Win95, or WinNT. The only difference is that, for Win 3.1 or Win95, you have to remove line

```
#define NT
```

from the file *c:\orgc\lib\envmsft.h*.

Compiling the library in the DOS window:

```
cd c:\orgc
```

msft
cd lib
msft

When using the VC++ debugger, all parts of the code must be compiled with the /Debug option. While debugging, compile orgC++ with lib\msftd.bat instead of lib\msft.bat.

When using the library, copy the environment file into your current directory:

cd ....\mydir
copy c:\orgc\lib\envmsft.h environ.h

In order to run the regression test:

cd c:\orgc\test
msftregr

ITEM 3: When running Visual C++ in the debugging mode (this is the most usual mode when developing new software), the compiler replaces the *new()* operator by an operator DEBUG_NEW. This happens for all classes that Class Wizard generated automatically for you. Since the Code Farm library always overloads operator *new()*, this results in a collision and compilation errors. In order to prevent this, comment the following line

#define new DEBUG_NEW

in all files generated by Microsoft that also use any classes registered under the Code Farm's library (classes that have ZZ_EXT_...). This disables the automatic memory leak detection provided by Visual C++, but there will be no conflict with the Code Farm's library.

ITEM 4: In Visual C++, the standard C preprocessor does not always behave as you would expect. In particular, *#define* and *#include* statements before and after *#include "stdafx.h"* have a different effect if you are using pre-compiled headers. Make sure that all the library defines and includes are after this statement.

#include "stdafx.h"
....
#define ZZmain
#include "zzincl.h"
#include "zzfunc.c"

ITEM 5: When you link your aplication, you may get warnings about *mllib.lib* not being compatible with VC++ runtime library. If this happens make sure that you compile the Code Farm's library with the same switches as your application.

BUILD - SETTINGS - C/C++: disable precompiled headers

BUILD - SETTINGS - LINK: add c:\orgc\lib\mllib.lib at the end

BUILD - SETTINGS - LINK: use /nodefaultlib:"libc.lib"

TOOLS - DIRECTORIES: add c:\orgc\lib

OPTIONS - DIRECTORIES: add c:\orgc and the directory of zzincl.h

2.5 Installing OrgC++ under OS2

Install and test the library using the same script files as for DOS, referring to the compiler you use (for example Borland).

No special script files are required.

2.6 Error 2104

The DOS version of the library uses a third party installation package, used by many other companies in the software industry. Some earlier versions of this software, when aborted in some irregular way, for example by rebooting the computer, left behind a working directory with one or more temporary files. On the next attempt to install the software, the installer stops with error 2104. This working directory may have been left on your disk by some other software which you tried to install sometime in the past.

In order to correct this situation, remove directory ISFYZQVO.TWJ and all the files that are in it. After that you will be able to install the library as expected.

« Chapter 1: Introduction

Chapter 3: Software Key »

3. SOFTWARE KEY

Prior to Ver. 3.0 , OrgC/C++ required a software key in order to run. There is no software key required now.

However, we would like to remind you that if you use this library without a proper license, or on more stations than you are entitled to, you are breaking the law, and become legally liable.

When you purchase a single copy of the library for a PC or Macintosh, you can keep a copy of the library on several different computers, for example in the office and at home, assuming that only one person is using the library at any given time.

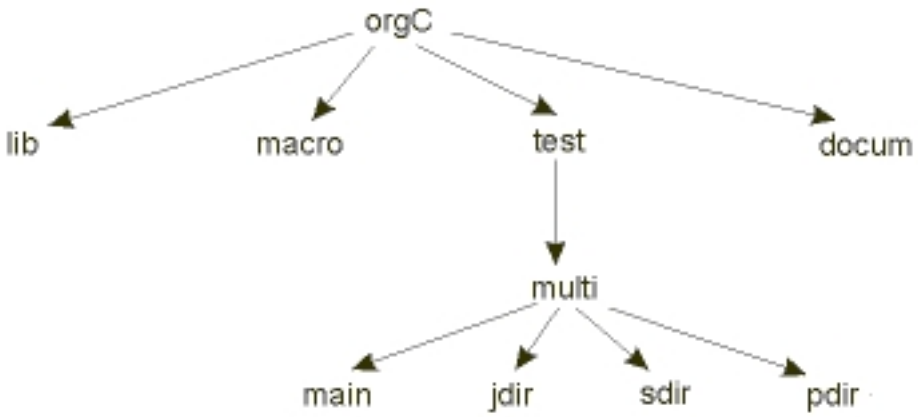
When you purchase a single copy of the library for a UNIX workstation, you can install the library on your workstation and on one PC or Macintosh station. Several users can work with the library on the same station, however, access by other users through a network is not permitted.

« Chapter 2: Installing Floppy or Tape

Chapter 4: Org C++ Directories »

4. DIRECTORIES

When you install OrgC/C++, you will have the following directories.



The *orgC* directory is the main OrgC++ directory, and it contains the preprocessor (*zzprep),* the help program *(zzhelp*), the file containing all of the combined macros (*zzcomb.h*), the program to combine macros (*zzcomb*), and a simplified *nroff* for DOS (*zzroff*).

The *lib* directory contains the OrgC/C++ library, to which you must link when you compile with OrgC/C++. Depending on the version of OrgC/C++ you are using, you may have a different library: *zzlib.a* (UNIX C),*zzclib.a* (UNIX C++), *zzlib.lib* (DOS TurboC medium data model), *zzllib.lib* (DOS TurboC large model), and so on.

The *macro* directory contains many files, one for each OrgC/C++ macro. Each file includes documentation, a simple example, and the executable code. You may examine these files either by using your favourite editor or, if you want to see it in better formatted text, call under UNIX:

nroff -me <filename>

or simplified *nroff* under DOS:

...\zzroff <filename>

The file *macro/zzmaster* is the most important file of the macro directory. It contains a master table of all organizations and functions available in the OrgC/C++ library, and must be properly updated if you add new features to the library (see Chap.16.)

The *docum* directory contains the program *zzdocum*, which automatically retrieves documentation from all macro files and puts it into the OrgC/C++ Reference Guide (*zzrefer*). *zzrefer* is in *nroff -me* format,

and it even includes new organizations and functions possibly added by the user. *zzdocum* can create either the complete documentation, or just update pages.

The *test* directory contains a large number of test programs that test every feature of OrgC/C++. The second half of Chap.5 describes how to invoke these tests.

The *multi* directory and all its subdirectories contain more complex examples, such as a project leader with 3 programmers using OrgC/C++ simultaneously at several levels (see Chap. 18).

« Chapter 3: Software Key

Chapter 5: Environment UNIX, DOS, MAC »

5. ENVIRONMENT: UNIX, DOS, MAC

OrgC/C++ is highly portable, and requires only minor changes when running in different environments. The file *environ.h*, which contains only several *#define* statements, specifies the hardware, operating system, and the compiler for the whole system. You cannot recompile the library without properly setting this file. If your *environ.h* file is incorrect, you will get numerous strange errors when compiling or trying to use the library.

Typically, you need a different *environ.h* file for the main *orgc* directory, and for *orgc/lib*. The class generator/preprocessor and utility programs are written in C (both the K&R and ANSI forms are available). The file in *orgc/lib/environ.h* specifies the environment under which you plan to use the library. If you plan to use the library with C, you have to recompile it under C. If you plan to use C++, you have to recompile the library under C++.

In the *environ.h* file, the first define specifies the operating system: *#define UNIX, #define DOS,* or *#define MAC.*

The second define specifies the hardware: *#define SUN, #define HP, #define AMDAHL, #define APOLLO, #define IBM (for RS6000).*

The third define specifies the compiler: *#define GNU, #define BORLAND4,* or *#define MICROSOFT*7. When running with C++, three more defines are usually used:

#define ZZcplus .... specifies C++
#define ZZansi .... specifies ANSI standard
#define ZZ_INHERIT .... inheritance or member object

When using large memory model under DOS, you define:

#define LARGE_POINTER

Chap.8 describes all of the available options in more detail.

When porting to a new compiler, the key decision is on how to paste tokens. Look at the file *lib/heading.h* and select your defines depending on which of the macros ZZ_PASTE1 ... to ZZ_PASTE4 works with your compiler. If you change *heading.h*, you have to rerun *zzcomb*, which recreates *zzcomb.h*.

The directory *orgc/lib* contains all library source files, plus numerous *env*.h* files for various possible environments. It also contains script files that compile the library under different environments. If you look into those files, you will see which *env*.h* files they use.

Knowing the naming conventions will help you orient yourself in this relatively busy directory. The libraries are named in the following fashion:

mllib	for Microsoft Visual C++
bmlib	for BorlandC++ medium memory model
bllib	for BorlandC++ large memory model
zzlib.a	for C on UNIX workstations
zgzlib.a	for UNIX GNU
zmlib	for ZORTECH C++ medium memory model
zllib	for ZORTECH C++ large memory model

File environ.h

The environment file describes the environment in which you are using the library: The operating system, physical computer, compiler, language (C or C++), memory model, and other parameters. Normally, the library administrator keeps a default file *environ.h* in the *orgc/lib* directory, which is then used by all programmers within the project or department. If you don't have the *environ.h* file in your current directory, all programs are automatically compiled with the default environment from *orgc/lib/environ.h*. However, if you have *environ.h* in your working directory, it overwrites the version in *orgc/lib*.

It is essential to have a correct *environ.h* file. If you have wrong defines, you get many strange compile errors. If you are starting with OrgC/C++ and you have a problem running the first test, first look at your *environ.h* files.

The directory *orgc/lib* contains many ready-to-use environment files; you have only to select the appropriate one and copy it into your current directory as *environ.h.* The names of these files relate to the environment for which they were designed. For example:

env0s.h	for C on SUN
env3z.h	for ZORTECH C++
env3g.h	for C++ using the GNU compiler
env3h.h	for the full HP C++
env3ca.h	for the latest version of BorlandC++
envmsft.h	for Microsoft Visual C++

and so on.

IMPORTANT: For WinNT or WinXP, add the following line to file lib\envmsft.h:
#define NT
For Win95 or Win98 remove this line if it is there.

If you run with SUN 2.1 (or 3.0) compiler, modify files *env0s.h, env1s.h, env2s.h,* and *env3s.h* by replacing *#define SUN* with *#define SUN2_1 (or #define SUN3_0).*

If you are not sure which file to use, look at *orgc/lib/readme* or at the script files for the regression tests (next paragraph).

Examples of the environ.h file:

For Microsoft Visual C++ under Win95:

#define DOS
#define MICROSOFT8
#define LARGE_POINTER
#define ZZcplus
#define ZZansi
#define ZZ_INHERIT

For GNU on a SUN, using C++:

#define UNIX
*#define SUN*3_0
#define GNU
#define GNUPLUS
#define ZZansi
#define ZZcplus
#define ZZ_INHERIT

For Borland C++ Ver.4.5, medium memory model:

#define DOS
#define BORLAND4
#define ZZcplus
#define ZZansi
#define ZZ_INHERIT

For ZORTECH C++, large memory model:

#define DOS
#define ZORTECH
#define LARGE_POINTER

#define ZZcplus
#define ZZansi
#define ZZ_INHERIT

For AT&T C++ Ver.2.1 on a SUN

#define UNIX
#define SUN2_1
#define ZZcplus
#define ZZansi
#define ZZ_INHERIT

If you are running with C++ but using structures and not classes, the following statment will make your program more efficient:

#define ZZnoFriends

Testing your setup.

Your new *orgc/test* directory contains examples of script files for running the preprocessor/class generator and for compiling a single program (files *try1* and *ctry2*, *gtry2*, or *htry2* for UNIX, and files such as *try1.bat*, *try2.bat*, *btry2.bat*, or *ztry2.bat* for DOS). Before using these files, check that (inside these files) the compiler is called with a path reflecting your individual installation.

- *Get the appropriate environment file*

Sun,C++	*cp ../lib/env3s.h*
Borland4,C	*copy ..\lib\env0ca.h environ.h*
Borland4,C++	*copy ..\lib\env3ca.h environ.h*
Visual C++	*copy ..\lib\envmsft.h environ.h*

- *Can you run the class generator?*

cd test
try1 test0m

- *Can you compile with OrgC++?*

Visual C++	mtry2 test0m
SUN,C++	ctry2 test0m
Borland,C++	:ctry2 test0m

Zortech	:ztry2 test0m
GNU	gtry2 test0m
HP C++	htry2 test0m

- *Are the results correct?*

UNIX	a.out inp0 res0m diff res0m out0mu
DOS/WIN	test0m inp0 res0m diff res0m out0ml

Important

The preprocessor/code generator must be called with full pass (absolute or relative). For example:

> *..\zzprep test0m.c*
> or
> *c:\orgc\zzprep test0m.c*
> or
> */home1/orgc/zzprep test0m.c*

If the path is not given *zzprep* cannot operate properly and prints a message about an internal error in function *ZZgetMaster()*. Having the *orgc* directory is not sufficient in this case.

Regression tests

The *orgc/test* directory contains scripts for long regression tests that run over 60 programs that test all of the features currently in the library. These tests are gradually expanding with the library.

The tests are different for C and C++, and also differ slightly between different compilers. For example, the UNIX tests sometimes use larger data sets than the DOS tests limited by the 64k data space, or some tests are excluded because the DOS compilers choke on them (not enough space).

These regression tests are the ultimate test of your installation, but may take a long time to run (1/2hour on SUN SPARC2, up to several hours on an IBM 386, but just a few minutes on a Pentium). Each script invokes one test after another, compiles and runs it, and then compares the result (file *res**) with the correct expected output (file *out**).

If you start a regression run, watch for the first couple of test passes. If everything looks fine, you may leave the test running without supervision and when it is finished, you invoke a special script file that checks the results:

msftregr	Visual C++ regression test
msftchk	Visual C++ checking results
cregr	SUN C++ regression test
ccheck	SUN checking C++ results

Expected output (files *out**) may sligthly differ for different compilers or memory models. For example, you may find:

out0m	usual result for *test0m*.c
out0mu	result for UNIX
out0ml	result for DOS large memory model

WARNING: Regression script files are quite long and, without a memory extender under DOS, their execution may be interrupted in the middle of a run because one of the DOS system limits is reached. If that happens to you, it still does not mean that your installation is wrong. Split the batch file into several sections, and run each separately.

Within each test run, file *environ.h* is reset several times. When splitting the batch file, make sure that *environ.h* is correctly set at the beginning of each section.

Examples of regression tests

cregr	UNIX, AT&T C++, SUN
gregr	UNIX, GNU, C++
mregr	MAC C++ (PWB only)
b4pregrm.bat	Borland 4.0, C++, medium memory model
b4pregrl.bat	Borland 4.0, C++, large memory model
msftregr.bat	Microsoft Visual C++ for WinNT
z3pregrm.bat	Zortech 3.0, C++, medium memory model
wtpregr.bat	Watcom 10.0+ 32 bit C++

Checking the results:

ccheck	UNIX, AT&T C++, SUN
mcheck	MAC C++
msftchk.bat	Microsoft Visual C++
b4pchkl.bat	Borland 4.0, C++, large memory model

... and so on (replace `regr' by `chk' or `check')

Warning:

The environment file must not contain conditional defines. For example, the following sequence will not do what you may expect:

```
#ifdef ABC
#define ZZansi
#endif
```

Regardless whether ABC is defined or not, the code generator will work as if *ZZansi* is defined.

« Chapter 4: OrgC++ Directories

Chapter 6: What is OrgC++? »

6. WHAT IS ORGANIZED C++?

OrgC++ is a package for the complete management of data in C++ programs, based on the new concept of *hyper-objects* (see below), sometimes also called *pattern classes*. OrgC++ is much more than just a class library. It provides an additional abstract layer, a whole subsystem for the management of data. You do not have to assemble the organization from library objects. The organization is automatically generated in a way which provides optimum run-time performance. Combined with automatic persistency (storage to disk) and version control, OrgC++ provides a set of powerful tools which are useful for a variety of tasks ranging from the management of internal data to the design of fast memory resident databases.

Compared to usual class libraries, OrgC++ has the following advantages: C

- Data persistency: Individual objects and whole organizations can be saved to disk in a single command, using either binary or ASCII format. Programmers do not have to code IO functions for every new class.
- Setting up the organization is much simpler. The user declares the organization in compact format, which is automatically translated into special transparent classes. There is no need to use inheritance, or to redeclare access functions, everything happens automatically.
- OrgC++ naturally handles complicated organizations which cause difficulties or inefficiencies in classical C++ libraries. One object may be part of many organizations, and one organization may include objects of several different types.
- Working with OrgC++ is like using database schema and access routines, but keeping the run-time performance of a custom coded program.
- Data portability between C and C++. Data created by a C program can be reopened by equivalent C++ programs.

Hyper-objects

What are hyper-objects? Hyper-objects are objects that store their data in other objects, called carriers. Carriers passively keep the data, but hyper-objects provide the methods. Hyper-objects are a perfect model for the organization of data in computer programs. For example, a graph can be represented as a hyper-object, which keeps its data (pointers forming the graph organization) in objects of type vertex and edge. Operations such as *"add"* or *"traverse"* are associated with the graph, not with vertices or edges.

Hyper-objects make the management of data conceptually clean. The new abstract layer removes unnecessary multiple inheritance/ friend declarations from the application code, and maintains optimum run-time performance.

There is a basic conceptual difference between OrgC++ and standard class libraries. For example, classical implementation of the linked list by Stroustrup uses two classes: *slink* which represents one link of the list, and *slist* which contains the entry into the list. When we split a list into two sublists, we need two instances of *slist*, but when we want to place an object into two parallel lists, we cannot easily differentiate between the two lists. Reference [5] in Appendix B explains this in a greater detail.

In OrgC++, when we split a list, the sublists, each with its own instance of *slink*, remains in the same hyper-object. Two parallel lists are represented as two instances of the list hyper-object.

Individual hyper-classes can be implemented in C++ with multiple inheritance and friend declarations, but an efficient library based on generic functions needs a special manager. OrgC++ supplies this missing link.

OrgC++ consists of a class generator and a special library. The class generator typically reads your header file with the declarations of your objects and organization, and it creates two files: an include file (the default name is *zzincl.h*), and a source file with all the access routines required by your program (default name *zzfunc.c*). You compile and link your original source code with these two files. This method has two advantages:

- Compiler and debugger messages refer to your original code.
- The class generator is called only when modifying data objects, not before every re-compilation.

OrgC++ provides a high level of error checking. As a result of strong typing, many coding errors are detected by the class generator or by the compiler. At run-time, OrgC++ protects your program against dangling pointers caused by incorrectly moving or deallocating some objects. It takes slightly longer to get a program through the compiler, but once it compiles, it is much safer to run.

The library contains all standard data objects:

- Singly or doubly linked rings, which can be also used as a list, ordered and unordered collection.
- Singly and doubly linked collections (linked lists with an encapsulated entry point).
- Object hierarchies and aggregations (different object type for each parent and for its children).
- Trees and general graphs (directed or undirected).
- Stacks and queues (LIFO, FIFO).
- Optional selfID tags (objects with this tag can report their own type).
- Optional timeStamp records the time when the object was created or modified.
- Run time extensibility of objects (Lisp property) allows you to expand objects by labelled attributes, which may be unknown at compile time.
- Dynamic arrays automatically expand when the index overflows the current size. The user controls whether and how the array will expand. Fast access without index checking is also available.
- Partially sorted binary heaps which control their sizes automatically.
- Hash tables depend upon two control functions: *compare_objects()* and *hash_name()*. OrgC++ provides default functions for regular users; sophisticated users may use their own functions.
- Four basic Entity-Relationship models are available: 1-to-1, 1-to-N, M-to-1, and M-to-N.

Most of the organizations come in source. The library is open. This means that users may modify the existing organizations and add new ones. All OrgC++ calls are fully encapsulated in C++ classes, and are separated from the application code. A special UTILITIES hyper-object contains general utilities such as allocation routines and saving to disk.

OrgC++ uses generic functions. For example, *add()* can be used to add an object to a ring, to a tree, or to a graph, without loosing the advantage of full type checking. The user does not have to remember many functions, only a small set of names such as *add(), del()* for deleting (disconnecting) an object, *fwd()* for moving forward in a chain of objects, or *par()* for reaching a parent in hierarchies or trees.

When creating a new object with a regular constructor, all internal data pointers are automatically initialized as "disconnected". When you insert *ZZ_CHECK()* into a class destructor, OrgC++ prevents objects from being deallocated prior to disconnecting pointers to other objects. Variable length names and other string-type

attributes are treated as objects.

Whole organizations of data can be saved to disk with a single command (structure blasting). When reading the data back into memory, all pointers forming the organization, and also all internal virtual function pointers are automatically restored. Binary or ASCII format can be used; the latter permitting the transfer of entire databases between different computers and operating systems.

Note that the C version of Organized C provides the same functionality in a less elegant setting. Hyper-objects there are declared in a very similar manner, but the access calls are macros instead of C++ methods.

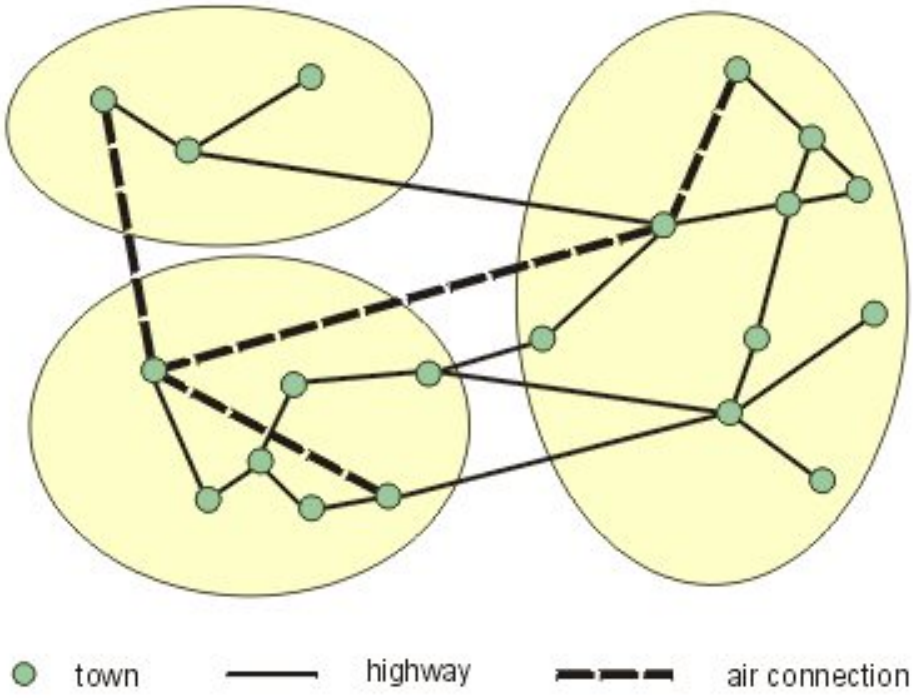
For example, the C++ situation

```
class listHead { ... };
class myObject { ... };
ZZ_HYPER_SINGLE_COLLECT(myList,listHead,myObject);
listHead *hp;
myObject *op;
hp=new listHead; // initialize empty list
...
op=new myObject;
myList.add(hp,op); // add op to myList
```

is, in C, represented as:

```
typedef struct listHead listHead;
typedef struct myObject myObject;
struct listHead { ... };
struct myObject { ... };
ZZ_ORG_SINGLE_COLLECT(myList,listHead,myObject);
listHead *hp;
myObject *op;
ZZ_PLAIN_ALLOC(listHead,1,hp);
...
ZZ_PLAIN_ALLOC(myObject,1,op);
ZZ_ADD(myList,hp,op); /* add op to myList */
```

One of the most important features of Organized C (both in C and C++) is that it unifies programming style between scientific (algorithmic) programming and databases. It allows organized database-like access to be used within algorithmic programs without losing the speed and efficiency of custom coded data and, on the other hand, allows rapid coding of complicated in-core databases which, when combined with hierarchical representation and saving to disk, work for essentially any data size.



Example of using OrgC++:

The problem involves towns located in different states. Two transportation networks connect the towns: highways and airline connections. The problem is to code a program, which will determine the least expensive route between a given pair of towns. The program will have several parts:

Part 1: General includes.

> *#define ZZmain // declares the main part of the program*
> *#include <<stdio.h>>*
> *#include "ZZinclude.h" // orgC++ class generator will create this file*

Part 2: Declaration of objects.

For each object, only the attributes that do not relate to the organization of data are declared. To make the example easy to read, these attributes such as *gasPrice* or *cost* are public even though, in a real application, they would most likely be private. Objects which will be involved in the automatic handling of data must contain a line of the form *ZZ_EXT_<objectType>*. This statement marks the place where the transparent pointers will be inserted.

```
class State {
    ZZ_EXT_State
public:
    float gasPrice;
};
class Town {
    ZZ_EXT_Town
public:
```

```
    float cost; // for the algorithm
    struct Town *trace; // route traceback
};
class Highway {
    ZZ_EXT_Highway
public:
    int distance;
};
class AirLink {
    ZZ_EXT_AirLink
public:
    float fare;
};
```

Part 3: Declaration of the organization.

Each line declares one organization as a hyper-class. For example, the first line declares a graph with *Towns* as nodes, and *AirLinks* as edges; *air* is the instance of the GRAPH organization. The words SINGLE and DOUBLE denote singly and doubly linked organizations. AGGREGATE is used when one object type contains several objects of another type. We always use RING to represent a linked-list (we do not recommend the use of NULL ending list). Performance and access are about the same, and RING allows better run-time error checking.

```
ZZ_HYPER_SINGLE_GRAPH(air, Town, AirLink);
ZZ_HYPER_SINGLE_GRAPH(hwy,Town, Highway);
ZZ_HYPER_DOUBLE_RING(states, State);
ZZ_HYPER_SINGLE_AGGREGATE(byState, State, Town);
ZZ_HYPER_UTILITIES(util);
```

Part 4: User program.

Once the organization has been declared in this manner, the program can manipulate the data by calling the functions associated with the hyper-objects. *states*, *byState*, *hwy*, and *air* are instances of organizations (hyper-objects). The *save()* function starts from a given object, called here "key entry", collects all objects connected to it by the network of pointers, and saves the whole organization to disk.

Often, data organizations require more than one key entry to collect all objects. This is the reason for having char *v[1],*t[1] which, in our simple case of one entry, may appear superfluous.

```
Highway *hp;
State *sp,*ss;
Town *tp;
char *v[1],*t[1];
tp=new Town; // allocates and initializes new town
byState.add(sp,tp); // adds town tp under state sp
hwy.del(hp); // deletes highway hp from graph "hwy"
ss=states.fwd(sp); // returns next state after sp
```

```
states_iterator sIter(ss); // starts iterator from ss
while(sp=sIter++){ // loops through all states
    printf("gasPrice=%f\n",sp->gasPrice);
}
v[0]=(char*)ss; /* entry into the data network */
t[0]="State"; /* type of the entry object */
util.save("myFile",v,t,1); // saves whole organization on file "myFile"
```

The iterators are automatically declared for all organizations specified in the *ZZ_HYPER_*.. statements. Typically, an iterator traverses the whole set (for example of a ring/list or collection), or a subset: for a tree it traverses the children of a given node, for a graph it traverses the adjacent edges and nodes.

The convenient form of the iterator *while(..=..){* is accepted by the AT&T C++ compiler, but ZORTECH and BORLAND compilers issue warning messages. If you want to avoid these messages, use the following statement which does the same thing, but is more difficult to read:

for(sp=sIter++; sp; sp=sIter++){ ...

Part 5: Changing the organization

Changing the organization is easy. For example, we may want to add a town name and a hash table that would allow fast selection of *Towns* by name. Also, the graph traversing algorithm may need a binary heap. All we need are the following additions:

```
class Header { // header for arrays and for the hash table
    ZZ_EXT_Header
};
ZZ_HYPER_NAME(townName,Town); // town name
ZZ_HYPER_ARRAY(heap,Header,Town*);// heap of town pointers
ZZ_HYPER_HASH(hash1,Header,Town); // hash table
```

Once this is declared, the new organizations can be used:

```
Header *hd;
Town t,*tp;
char *cp;
int cmpFun(char *,char *); // compares two objects as for qsort()
cp=util.str("name1"); // allocates new string "name1"
townName.add(&t,cp); // adds name to temporary town t
...
p=hash1.get(hd,&t); // finds Town with this name
if(tp){
    cp=townName.fwd(tp); // returns name of town tp
    printf("town=%s\n",cp);
    heap.inHeap(cmpFun,hd,tp); // inserts tp into the heap
    }
```

Important:

If you allocate all objects with operator *new()*, as in the example above, pointers in all objects are automatically initialized as disconnected (*NULL*). Without a proper initialization, you are likely to run into serious problems with the data.

Objects are not automatically initialized, if you allocate them automatically. In such a case, you must include macro *ZZ_INIT()* in the constructor of the class. For example if, instead of what we had above:

```
    class Town {
        ZZ_EXT_Town
    public:
        float cost;
        Town *trace;
    };
    Town *tp;
    tp=new Town;
    ...
```

you write:

```
    class Town {
        ZZ_EXT_Town
    public:
        float cost;
        Town *trace;
        Town(){ZZ_INIT(Town);}
    };
    Town tp;
    ...
```

It is recommended to be consistent and use the same style for all your classes: Either use *ZZ_INIT()* in all your constructors, or avoid automatically allocated objects.

« Chapter 5: Environment UNIX, DOS, MAC

Chapter 7: Examples »

7. EXAMPLES

7.1 COMPLETE PROGRAM WITH ORGC++

This chapter contains a simple but complete program with OrgC++. Comments have been included to explain each step. If you are a new user, read this program carefully. You may also type it into a computer and try to modify it (if you type every statement, you'll notice things that you may miss if you just read the existing code). This program is a C++ version of the program *test15a.c*, available on the distribution disk. If you understand this example, you understand the essence of OrgC++.

The task is to read a list of employee records, link the records into two lists, sort one alphabetically and the other by salary, and then print both lists. Input format: *emplName salary*.

In OrgC++, rings are always used instead of NULL ending lists. Operations on both organizations are very similar, but rings allow better protection against run-time errors.

The OrgC++ library provides two types of rings:

- RING is just a plain ring with an outside pointer, which is equivalent to the head/tail of the list, and which must be externally managed;
- COLLECTION is a ring with its entry point encapsulated in another object.

In the following program, the COLLECTION is used. The *Header* object encapsulates the entry points for both lists.

```
#include <stdio.h> // usual io include
#define ZZmain // main file for OrgC++ global variables
#include "zzincl" // OrgC++ will generate this file
class Header { // header for both lists
    ZZ_EXT_Header // automat.controlled pointers
public:
};
class Employee { // employee record
    ZZ_EXT_Employee // automat.controlled pointers
public:
    static int sfun(const void*,const void*);
    static int nfun(const void*,const void*);
```

```
    char *name;
    int salary;
};
ZZ_HYPER_SINGLE_COLLECT(bySalary,Header,Employee);
ZZ_HYPER_SINGLE_COLLECT(byName,Header,Employee);
ZZ_HYPER_UTILITIES(util);
#define BSIZE 80
main(int argc, char **argv){
    char name[BSIZE],buff[BSIZE],*nn;
    FILE *file1,*file2;
    Header *hp; // header for both lists
    Employee *ep; //just a pointerT
    int sal;
    if(argc<=1)file1=stdin; else file1=fopen(argv[1],"r");
        if(argc<=2)file2=stdout; else file2=fopen(argv[2],"w");A
    hp=new Header; // automatically empty lists
    while(fgets(buff,BSIZE,file1)){
        sscanf(buff,"%s %d",name,&sal);
        ep=new Employee; // get new Employee object
        nn=util.strAlloc(name); // allocate and copy name
        ep->name=nn;
        ep->salary=sal;
        bySalary.add(hp,ep); // add to one list
        byName.add(hp,ep); // add to the second list
    } // all input loaded in
    bySalary.sort(Employee::sfun,hp); // sort one list
    byName.sort(Employee::nfun,hp); // sort the second list
    bySalary_iterator sIter(hp);
    while(ep=sIter++){ // traverse list bySalary
        fprintf(file2,"%s %d\n",ep->name,ep->salary);
    }
    fprintf(file2,"\n");
    byName_iterator nIter(hp);
    while(ep=nIter++){ // traverse list byName
        fprintf(file2,"%s %d\n",ep->>name,ep->>salary);
    }
}
// Compare functions are similar to qsort().
// Return: -1 for p1<p2,0 for p1==p2,+1 for p1>p2
int Employee::sfun(const void *p1, const void *p2){
    Employee *e1,*e2;
    e1=(Employee *)p1; e2=(Employee *)p2;
    if(e1->salary<e2->salary)return(-1);
```

```
    if(e1->salary>e2->salary)return(1);
    return(0);
}
int Employee::nfun(const void *p1, const void *p2){
    Employee *e1,*e2;
    e1=(Employee *)p1; e2=(Employee *)p2;
    return(strcmp(e1->name,e2->name));
}
#include R zzfunc.cR // class generator creates this file.
```

We assume that we are running on PC DOS with BorlandC++ 3.0, Organized C++ has been installed in the file *c:\orgC* , and the program and input file are both in the directory *c:\test\myDir*, called *test.cpp* and *inp*. BorlandC++ has been installed in *c:\bc*, and we are using the medium memory model.

First we run the class generator: *try1 test*
Then we compile and link: *btry2 test*
And then we execute the program: *test inp out*

Input (file *inp*)

White, J.	1100
Brown,G.	850
Jones,C.	200
Green,F.	900
Jones,S.	450
Black,R.	790
Moody,S	950

Result of the run (file *out*):

Jones, C.	200
Jones, S.	450
Black, R.	790
Brown, G.	850
Green, F.	900
Moody, S.	950

White, J.	1100
Black, R.	790
Brown, G.	850
Green, F.	900
Jones, C.	200
Jones, S	450
Moody, S.	950
White, J.	1100

Comment:

Files: *try1.bat* and *btry2.bat* below show how to invoke the class generator, and how to compile and link.

try1.bat: c:\orgC\zzprep %1.cpp
try2.bat: bcc -mm -Vt -Lc:\bc\lib -Ic:\bc\include %1.cpp c:\orgC\lib\bmlib.lib

7.2 CONVERTING REAL DATA INTO ORGC++ ORGANIZATIONS

Programming with OrgC++ enforces the object oriented style of programming. Before you start to code, you have to plan not only the logic of the program, but also the organization of the data.

The relations between data, usually represented as a network of pointers, must be reduced to standard data sets such as linked lists, trees, or hash tables.

Depending on your personal background, this can be done in three different ways:

- If your background is purely object-oriented, you may want to describe your data as associations and aggregations, and then map these relations into the OrgC++ library. Note that the same association may be implemented in a number of different ways. For example, an aggregation can be implemented as COLLECT, AGGREGATE, or HASH in OrgC++.
- If you have some experience with pointer implementation in data structures such as linked lists, you may want to draw the pointer network, and derive the organization from the picture. You have to decompose the general network into simple subsets available from the library, which is usually quite simple.
- If you are not an experienced programmer, and you know very little about data structures, the whole notion of data representation may be confusing to you. Simply start to code as if there are no data structures and no OrgC++ library, but when you reach the situation that you need to add a pointer to one of the classes, stop and think why you need that pointer. That will naturally lead you

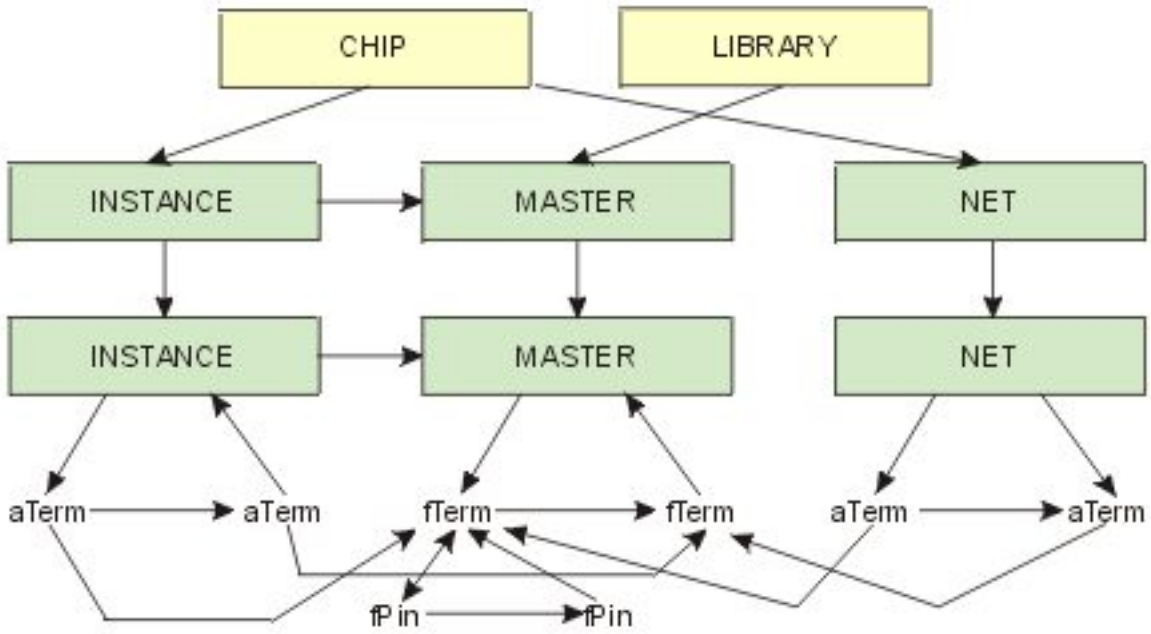
to one of the diagrams shown in Chap.11. The rule is simply that your classes must not contain any pointers.

Our experience is that most users of OrgC++ libraries fall into the second (middle) category. For that reason, we will start from pointer diagrams in the two examples below.

Example 1:

An electrical netlist (VLSI chip or printed board layout) involves the following entities. A library contains masters of cells; each cell has formal terminals; each formal terminal is a logical entity; connections to a formal terminal may enter the cell through one or more physical pins. The whole chip, and each block of the layout is composed of instances of cells connected by signal nets. A cell has actual terminals (which are instances of formal terminals), and actual pins (which are instances of formal pins). A signal net connects one or more actual terminals.

The following diagram not only describes the data relations; it also contains the decision about which traversals require fast access. For example, if you plan to build the layout and gradually add new cells, a singly-linked list of cells is the best solution. If you plan to delete cells often, a doubly-linked list works better.



The two-sided relation between cell and aTerm, and between net and aTerm is sometimes called a netlist, and is the logical skeleton of the whole organization.

Pins are the most abundant object here, and if not handled carefully, they can consume a lot of memory. For this reason, actual pins are usually not stored, and are only derived. For a given aTerm, the program can quickly get its corresponding fTerm, and from there all its formal pins. The position of the pins is

derived on-the-fly by combining the position and rotation of the cell with the coordinates of the pin within the master cell.

Usually, when deciding on an OrgC++ organization, we try to extract the largest organizations first. In this example, there are no trees or graphs, only several aggregates (formerly called triangles, see Chap.11.3). These aggregates are: aTerms under Instance, aTerms under Net, and fTerms under Master. The paths from a fPin to the fTerm and from a net to the block are usually not required; we use a collection, which is less memory expensive than an aggregate, because it does not require the parent pointer.

The whole organization can be declared in several lines:

```
ZZ_HYPER_SINGLE_COLLECT(inst,Chip,Instance);
ZZ_HYPER_SINGLE_COLLECT(nets,Chip,Net);
ZZ_HYPER_SINGLE_COLLECT(lib,Library,Master);
ZZ_HYPER_SINGLE_AGGREGATE(terms,Master,fTerm);
ZZ_HYPER_SINGLE_AGGREGATE(byInst,Instance,aTerm);
ZZ_HYPER_SINGLE_AGGREGATE(byNet,Net,aTerm);
ZZ_HYPER_SINGLE_COLLECT(pins,fTerm,fPin);
ZZ_HYPER_SINGLE_LINK(iLink,Instance,Master);
ZZ_HYPER_SINGLE_LINK(tLink,aTerm,fTerm);
```

Example 2:

This example involves a collection of real-time data of the eye positions of people watching a television program.

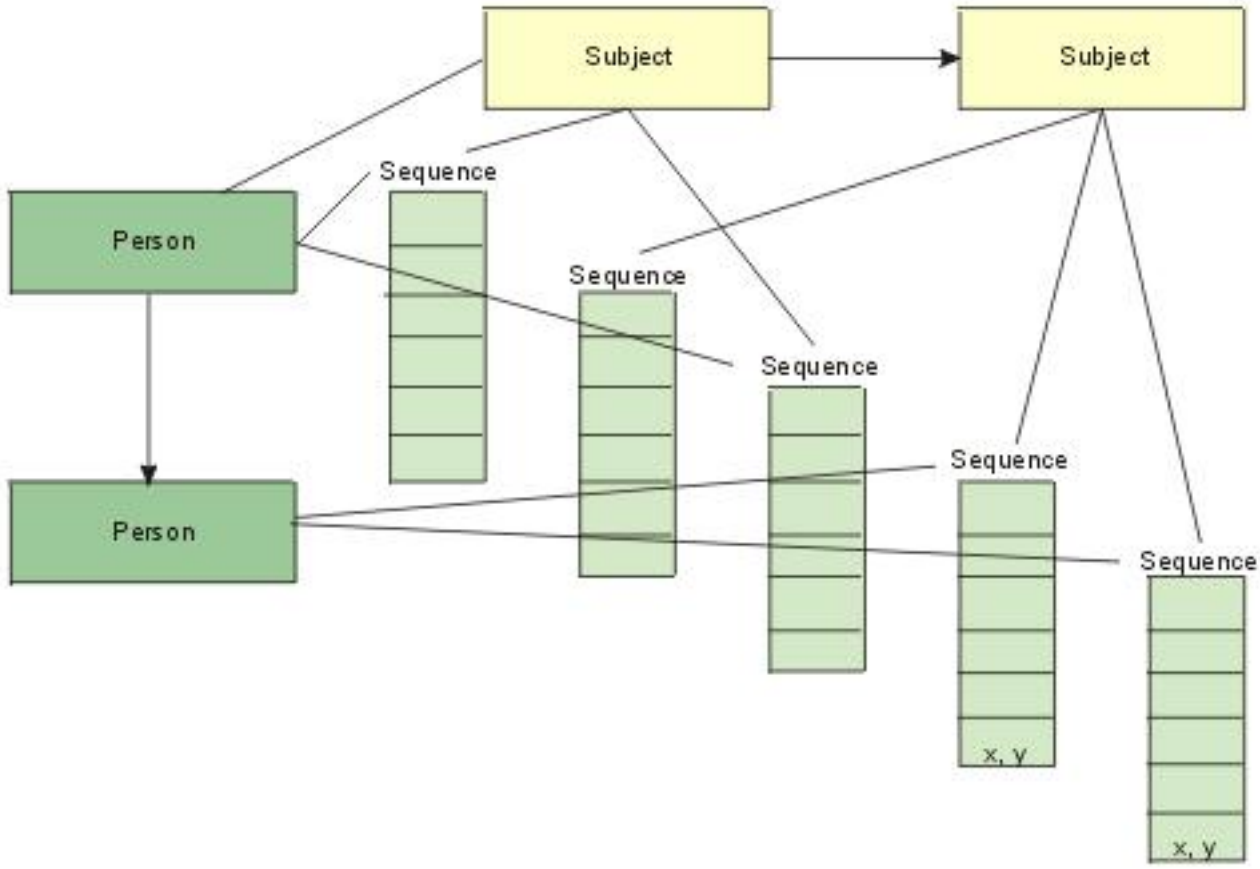
The data relates to several video tapes and several people. One experiment consists of recording a sequence of x,y coordinates from one viewing, and should provide easy comparison of recordings for the same subject and a different person.

There are many xy coordinates in each sequence, and storing them in an array saves memory. Each sequence has a different number of recorded positions (the length of observation is somewhat random), therefore a dynamic array is required. The diagram below shows a sketch, such as one encounters when discussing a new project.

When designing the data organization, we have to consider which relations are most frequently accessed:

- for each sequence, access to subject and person;
- for each person, access to all sequences;
- for each subject, access to all sequences;
- each sequence contains an array of eye positions;
- persons and subjects require linear access (a linked list or a ring);

- person and subject have a name.



This description translates into the following OrgC++ declaration:

```
ZZ_HYPER_SINGLE_RING(pRing,Person);
ZZ_HYPER_SINGLE_RING(sRing,Subject);
ZZ_HYPER_SINGLE_AGGREGATE(byPerson,Person,Sequence);
ZZ_HYPER_SINGLE_AGGREGATE(bySubject,Subject,Sequence);
ZZ_HYPER_ARRAY(positions,Sequence,Eye);
ZZ_HYPER_NAME(pName,Person);
ZZ_HYPER_NAME(sName,Subject);
```

7.3 SAVING TO DISK AND VIRTUAL FUNCTIONS

Naturally, when programming in C++, one of the most important questions is how to save objects when virtual functions are used (persistency). With virtual functions, pointers forming the data organization often connect only to the object from which other objects are derived. When traversing or saving the data the programmer does not have direct access to the derived objects. The following program is a typical example. If you want to try or modify this example, the source code is in *orgC/test/test23a.c*.

In this example, instead of using the global *util.save()* function, which collects all objects and saves them,

we traverse all base objects and save the derived objects using the virtual function *save()*.

The program forms a ring of Shapes, where each Shape can be either a Square or a Rectangle. The whole organization is saved on disk using ASCII format, then brought back into memory, saved again for testing purposes, this time in the binary format, then retrieved again, and the result is printed. In both input/output a Square is described by one side, a Rectangle with two sides.

```
#include <stdio.h>
#define ZZ_INHERIT
#define ZZmain
#define ZZasci
#include "zzincl.h"

class Root
{
    ZZ_EXT_Root // OrgC++ pointers are private
public:
    void save(char *);
};
ZZ_FORMAT(Root,""); // needed for ASCII save only
class Shape {
    ZZ_EXT_Shape
public
    virtual void prt(void){};
    virtual void save(char *){};
};
ZZ_FORMAT(Shape,"");
class Square : public Shape{
    int x;
    ZZ_EXT_Square
public:
    void prt(void){printf(R Square %d\nR ,x);};
    void set(int a){x=a;};
    virtual void save(char *file);
};
ZZ_FORMAT(Square,"%d,x"); // all classes with ZZ_EXT_..
class Rectangle : public Shape{
    int x,y;
    ZZ_EXT_Rectangle
public:
    void prt(void){printf("Rectangle %dx%d\n",x,y);};
    void set(int a,int b){x=a; y=b;};
    virtual void save(char *file);
```

```
};
ZZ_FORMAT(Rectangle,"%d %d,x,y");
void Root::save(char *file){ZZ_STORE(Root,file);};
void Square::save(char *file){ZZ_STORE(Square,file);};
void Rectangle::save(char *file){ZZ_STORE(Rectangle,file);};
ZZ_HYPER_SINGLE_COLLECT(all,Root,Shape); // Root contains Shapes
ZZ_HYPER_UTILITIES(util); // needed for save() and open()
#define BF 80
char buff[BF];
int main(void)
{
    char c[20],*v[1],*t[1];;
    int x,y;
    Root *rt;
    Square *sp;
    Rectangle *rp;
    Shape *pp;
    void prtData(Root *);
    rt=new Root; // all data under one root
    while(gets(buff)){ // read until the EOF
        sscanf(buff,"%c",c);
        switch(c[0]) {
            case 'S':
                sp=new(Square);
                sscanf(buff,"%c %d",&c,&x);
                sp->set(x);
                all.add(rt,(Shape *)sp);
                break;
            case 'R':
                rp=new(Rectangle);
                sscanf(buff,"%c %d %d",&c,&x,&y);
                rp->set(x,y);
                all.add(rt,(Shape *)rp);
                break;
            default: printf("wrong input code\n");
            }
        }
        prtData(rt); // print all data as loaded
        rt->save("afile"); // save the root
        all_iterator it(rt); // initialize iterator
        while(pp=it++){ // traverse the collection
            pp->save(R ); // save each object
        }
```

```
    util.close("afile"); // close the file
    util.open("afile",1,v,t); // retrieve data from "afile"
    rt=(Root *)v[0]; // recover key entry
    prtData(rt); // prints the second copy now in memory
    util.mode(0,1,0,0); // switch to binary format
    rt->save("bfile"); // save the root
    it.start(rt); // re-start the iterator
    while(pp=it++){ // traverse the collection
        pp->save("bfile"); // save each object
    }
    util.close("bfile"); // close the file
    util.open("bfile",1,v,t); // retrieve data from bfile
    rt=(Root *)v[0];
    prtData(rt); // prints the third copy now in memory
    return(0);
}
//_____________________________________________________________________
void prtData(Root *rt) //print out all data
{
    Shape *pp;
    all_iterator it(rt);
    while(pp=it++){ pp->prt(); }
}
#include "zzfunc.c"
```

INPUT DATA:

```
S 5
S 8
R 2 6
R 3 1
S 7
```

COMMENT:

Instead of saving individual objects, as shown above, the entire data can be saved in one command:

```
v[0] = (char*)rt; t[0] ="Root";
util.save("afile",1,v,t);
```

OrgC++ saving utility is intelligent enough to traverse the collection of shapes, to detect the type of each shape, and to invoke the appropriate saving function automatically.

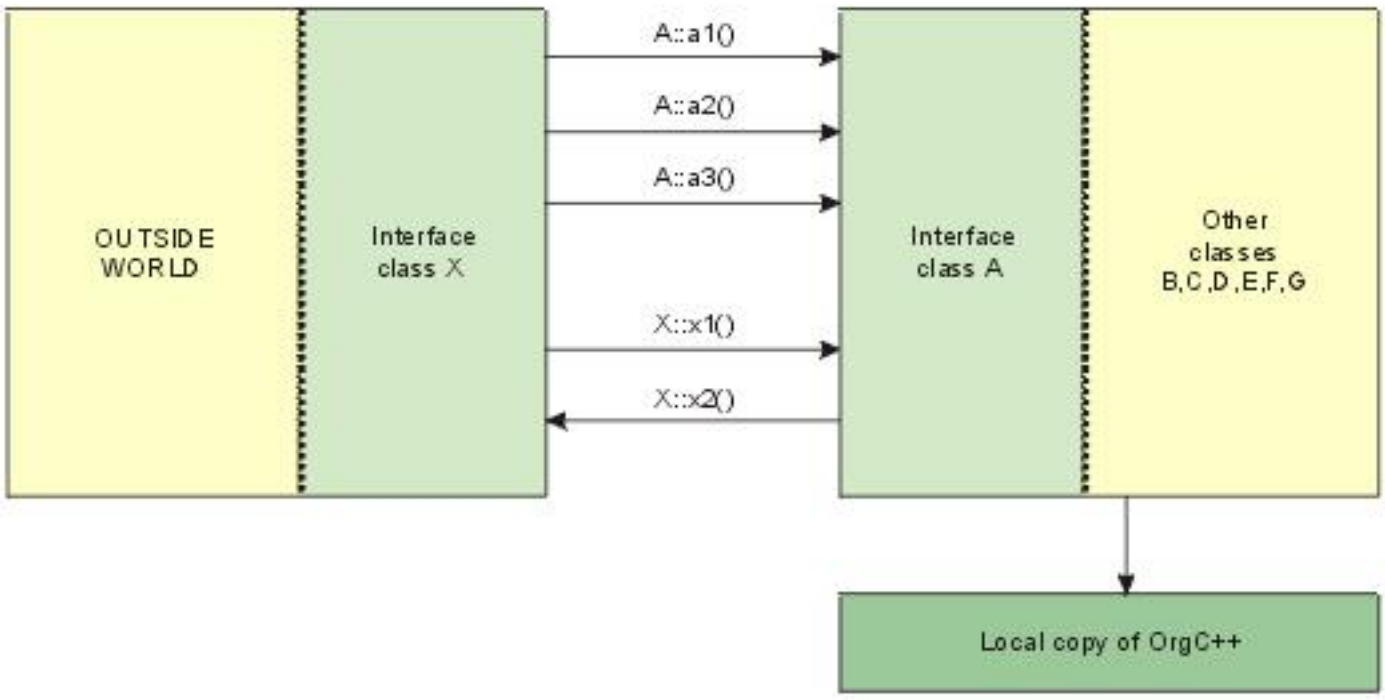
IMPORTANT:

Read the comment about initialization of objects and the use of *ZZ_INIT()* on Chap 13.2. This macro must be in all constructors, if you use automatically allocated objects.

7.4 ADVANCED EXAMPLE: DLL under WindowsNT

The purpose of this example is to demonstrate the overall organization and settings needed for the VisualC++ environment, without going into details of the actual coding. This is not an introductory example - it assumes you already know quite a bit about OrgC++, and you are using the Microsoft Development Studio with the Visual C++ under Windows NT.

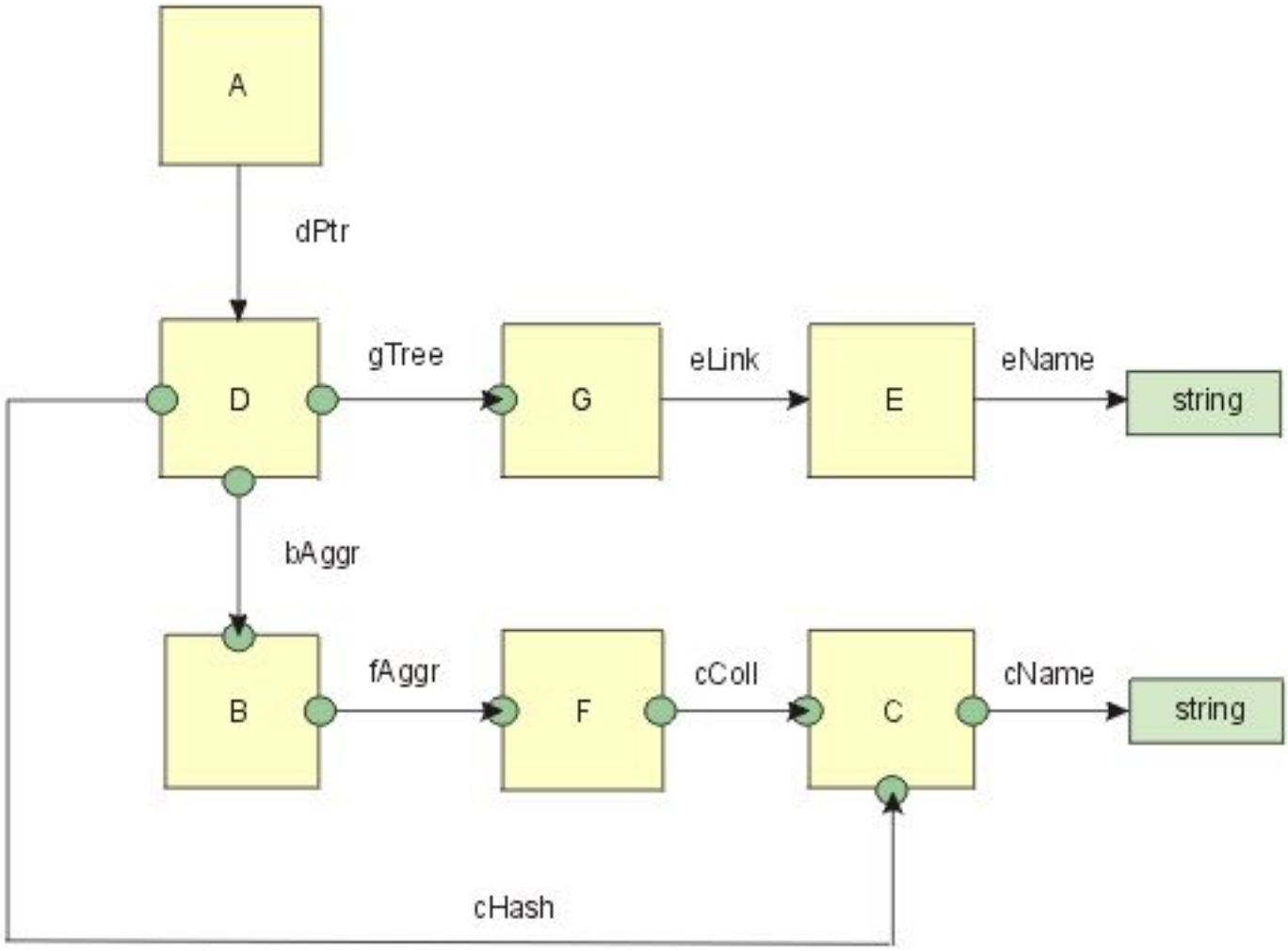
We have the problem as shown in the following diagram. The program module with classes A,B,..,G uses the OrgC++ library, and should be implemented as DLL (dynamic link library). Class A provides the external interface through functions *A::a1(), A::a2(), A::a3()*. For each of the library classes, we will have two usual files, **.h* and **.cpp*. The outside world does not know about classes B,C,...,G; it knows only A.h. This DLL (we will call it *mylib*) uses its own, local copy of OrgC++, which must not interfere with other parts of the system that may also use OrgC++.



Also, in this case, the library must call some external functions, for example when reporting errors or special events. Let us assume that these external functions are methods of class X:

Classes A,B,...,G form the organization shown in the next figure. This diagram matches the *ZZ_HYPER_*.. statements recorded in file *hyper.h* which is a part of the *mylib* project.

To make this example more realistic, let us assume that *mylib* may need to save/restore its internal data from disk, and let us place this functionality into *A::a1()* and *A::a2()*. Note that we will not save the instance of A which is not an OrgC++ registered class. It does not have the *ZZ_EXT_A* statement because class X does not know about OrgC++ and file *zzincl.h* . The pointer between A and D is just a regular pointer, not a *ZZ_SINGLE_LINK*.



Here is the file *hyper.h* , which also includes the functions required for the hashing:

```
ZZ_HYPER_HASH(cHash,D,C);
ZZ_HYPER_SINGLE_AGGREGATE(bAggr,D,B);
ZZ_HYPER_SINGLE_AGGREGATE(fAggr,B,F);
ZZ_HYPER_DOUBLE_COLLECT(cColl,F,C);
ZZ_HYPER_SINGLE_TREE(gTree,D,G);
ZZ_HYPER_SINGLE_LINK(eLink,G,E);
ZZ_HYPER_NAME(cName,C);
ZZ_HYPER_NAME(eName,E);
```

ZZ_HYPER_UTILITIES(util);

```
//-------------------------------------------------
//              A
//              |ptr
//              v  gTree        eLink   eName
//      +-----o D o-----x G ----- >E ----- >string
//      |       o
//      |       |bAggr
//      |       |
//      |       x  fAggr        cColl   cName
//      |       B o-----x F o---- x C ----- >string
//      |                           x
//      |               cHash       |
//      +---------------------------+
//
//-------------------------------------------------
```

*inline int cHash_class::cmp(C *c1,C *c2){...}*
*inline int cHash_class::hash(C *c, int size){...}*

In addition to files *A.cpp, B.cpp, ... , G.cpp*, you also need file *func.cpp*, which represents the 'main' file for OrgC++, and includes file *zzfunc.c* generated by the OrgC++ code generator.

File *func.cpp*:

```
#include <stdio.h>
#include <string.h>
#define ZZmain
#include "zzincl.h"
#include "A.h"
#include "B.h"
#include "C.h"
#include "D.h"
#include "E.h"
#include "F.h"
#include "G.h"
#include "zzfunc.c"
```

Implementation of each class depends only on the classes and *ZZ_HYPER_*.. statements that relate directly to each class. For example, for class G, we have file *G.h*:

```
class G {
```

```
    ZZ_EXT_G
public:
    G();
    int fun(char *c);
    ...
};
ZZ_FORMAT(G,"..."); // only when ASCII saving to disk
```

File *G.cpp* depends on files *D.h*, *E.h*, and *G.h*:

```
#include <stdio.h>
#include <string.h>
#include "zzincl.h"
#include "D.h"
#include "E.h"
#include "F.h"
ZZ_HYPER_SINGLE_TREE(gTree,D,G);
ZZ_HYPER_SINGLE_LINK(eLink,G,E);
G::G(){ ... }
int G::fun(char *c){ ... }
```

In order to invoke the OrgC++ code generator, you will need a special batch file which we will call *generate.bat* :

```
copy c:\orgc\lib\envmsft.h environ.h
copy A.h all.inc
type B.h > all.inc
type C.h > all.inc
type D.h > all.inc
type E.h > all.inc
type F.h > all.inc
type G.h > all.inc
type hyper.h > all.inc
c:\orgc\zzprep all.inc
```

The definition of classes A and X requires an intricate definition of how to communicate with the other part of the project. Class A keeps a pointer to each of the external functions:

File A.h:

```
#ifdef A_SIDE
#define A_EXPORT __declspec(dllexport)
#else
```

```
#define A_EXPORT
#endif

class A_EXPORT A {
typedef void (X::*x1Type)();
typedef int (X::*x2Type)(int);

x1Type x1Fun;
x2Type x2Fun;
D *dPtr;
X *xPtr;
public:
A(X *xp, x1Type x1f,x2Type x2f);
void a1(char *fileName); // save data
void a2(char *fileName); // restore data
int a3(); // whatever
...
};
```

File A.cpp:

```
#define A_SIDE
#include <stdio.h>
#include <string.h>
#include "zzincl.h"
#include "A.h"
#include "X.h"
ZZ_HYPER_UTILITIES(util);
A::A(X *xp, x1Type x1f, x2Type x2f){
    xPtr=x;
    x1Fun=x1f;
    x2Fun=x2f;
    ...
}
void a1(char *fileName){
    void *v[1]; char *t[1];
    v[0]=dPtr; t[0]="D";
    util.save(fileName,1,v,t);
}
void a2(char *fileName){
    void *v[1]; char *t[1];
    util.open(fileName,1,v,t);
    if(!util.error() &&
```

```
        !strcmp("D",t[0]))dPtr=(D*)(v[0]);
        else (xPtr-*x1Fun)();
    }
```

Note the use of *X::x1()* in the last statement of the implementation of *A::a2()*!!

Since X.h is needed for class A, its definition also has to reflect the arrangements about the interface. Here is file *X.h*:

```
    #ifdef A_SIDE
    #define X_EXPORT __declspec(dllimport)
    #else
    #define X_EXPORT
    #endif
    class X_EXPORT X {
        ...
    public
        void x1();
        int x2(int i);
        ...
    };
```

And file *X.cpp*:

```
    #include "X.h"
    #include "A.h"

    void X::x1(){ ... }
    int X::x2(int i){ ... }
```

Here are the suggested Build/Settings values for this situation. Don't forget that for Debug/Release options, you have to link to a different version *mllib* (see Chap.2.4).

General: Nothing special
Debug: Nothing special
Custom Build: Nothing special (see below)
C/C++: Precompiled headers, not using them
Link: Object/library modules: include *c:\orgc\lib\mllib.lib*
    Input: ignore libraries: *libcd.lib*

Microsoft Developer Studio offers `Custom Build' and `Preprocessor' options, but these options seem to fit only primitive preprocessors. We have not succeeded to apply them to the OrgC++ code generator, *zzprep*. We found that it does not cause much difficulties to call *generate.bat* manually (either from `run'

or from the DOS window) when modifying any *ZZ_HYPER_*.. statement or class members in any of the **.h* files.

« Chapter 6: What is OrgC++?

Chapter 8: Syntax »

8. SYNTAX

8.1 INCLUDES AND DEFINES

8.1.1 ZZmain

DOL uses only a few global variables, and *#define ZZmain* designates the file where these variables will be declared. This often is the file with the *main()* program, but it does not have to be. Simply select one of your files to be the *main* DOL file. If your program is all in one file, you need *#define ZZmain*.

If several programmers use DOL in a hierarchical fashion, as described in Chap.18, only one file can have *ZZmain*. Using *ZZmain* in more than one file will cause the linker to complain about multiply declared variables.

8.1.2 *zzincl.h* and *zzfunc.c*

The DOL class generator creates two files, with which you have to compile your source. The default names for these files are *zzincl.h* and *zzfunc.c.* Your program usually starts with *#include "zzincl.h.*", and has *#include "zzfunc.c"* at the end of one of your files, or compiles *zzfunc.c* separately.

Note that *zzincl.h* also contains, besides other include directives, the statement *#include "../zzcomb.h."*, which includes the whole macro library.

zzincl.h (not *zzfunc.c*) is protected against multiple inclusion. Simply, you can use it everywhere without worrying about multiple declarations.

If you want to choose different names for these files, avoid using the prefix ZZ, which is reserved for DOL variables and functions. The general syntax for calling the class generator is:

orgCpath/zzprep prog.c <incl.h> <zzmaster> <func.c>

where prog.c is the file with your program, and *incl.h* and *func.c* are the files equivalent to *zzincl.h* and *zzfunc.c*, and *master* is the master-control file of your macro library, usually *zzmaster*.

zzprep must be given a full or relative path to the orgC directory; other files are given without the path. *incl.h* and *func.c* will automatically be in the current directory, *master* in the macro directory under the directory where *zzprep* is (typically *orgC/macro*).

If your program consists of several files, you have two options:

Either run the class generator only on your header file, which contains the declarations of all objects and their organizations, or concatenate all your files together and run them through the class generator.

File *zzincl.h* automatically pulls in *stdio.h*. Most examples in the test directory include *stdio.h* explicitely, as it was needed in OrgC/C++ prior to version 3.0. Today it is not necessary.

If you use VisualC++ from Microsoft, note that file *zzincl.h* must be included after *stdafx.h:*

```
#include <stdafx.>
#include "zzincl.h"
```

8.1.3 ZZimplicit

ZZimplicit is one of the important defines in the C version of the library (Data Manager for C, OrgC), but it is not used in DOL.

8.1.4 ZZselectMacros

When delivering a finished software product, it is not necessary to send the whole macro library (*zzcomb.h*) with it. If *#define ZZselectMacros* is used, the class generator selects those macros that are actually used by your program, and places them into the file *zzcomb.h* in your current(!) directory. File *zzincl.h* is then instructed to pick up that file, and not *zzcomb.h* from the main OrgC directory.

ZZselectMacros causes a relatively long class generator run, makes shorter include files, and slightly shorter compilation, with no effect on run-time. It is a good idea to develop a program without this *#define*, and add it only just before compiling the final version of the software.

8.1.6. ZZascii

Normally, when saving data to disk, binary format is used. For ASCII format, (also see Chap.13.2), use *#define ZZascii*. This define must also be used when using both formats within the same program - see macro *ZZ_MODE_SAVE()*. As an example, compare *test16a.c* (saving in binary format) with *test16b.c* (saving in ASCII format).

When using the interactive browser, this define must be used, because the browser is printing objects in the ASCII format on the screen.

8.1.7. ZZcplus

When running with C++, *#define ZZcplus* must always be present in your *lib/environ.h* file. For example, when using ZORTECH C++ the file should contain, and possibly other defines, the following statements:

#define DOS
#define ZORTECH
#define ZZcplus

The class generator automatically creates necessary friend declarations and inserts them under the appropriate *ZZ_EXT* statements. The generation of friends may be disabled by using

#define ZZnoFriends.

This may be useful if running C++ in C-like style, with structures instead of classes.

If you are running with C++ Ver.2.1 or higher, declare *ZZcplus21* instead of *ZZcplus*. For example, when running C++ Ver.3.0 on Sun, you need:

#define UNIX
#define SUN3_0
#define ZZcplus21
#define ZZansi
#define ZZ_INHERIT

8.1.8 ZZ_INHERIT

This define is required when using DOL with hierarchical types, which means in most situations when using C++. The only exception is when using light-weight objects (no inheritance) and no member objects:

```
class A {
    ...
};

class B {
    A a; // should not be used
    A *ap; // this is OK
    int i; // no problem
    ...
};
```

Hierarchical types can occur in two ways: through C++ inheritance or through a class member object. Note that the second situation can occur even in regular C.

ZZ_INHERIT is also required, if some pointers lead into the middle of an array, or when using *save()/open()*on objects with virtual functions (if any of the objects with *ZZ_EXT_*.. use virtual functions). For more details on pointers that lead into an array- see ARRAY (Chap.11.12).

CASE 1:

Hierarchical types through inheritance.

```
class myRing {
    ZZ_EXT_myRing
public:
    ...
};
class box: myRing {
    ZZ_EXT_box
public:
    ...
};
```

CASE 2:

Hierarchical types through a member class object.

```
class Obj1 {
    ZZ_EXT_Obj1
    ...
};
class Obj2 {
    ZZ_EXT_Obj2
public:
    Obj1 a;
    ...
};
ZZ_HYPER_SINGLE_RING(ring1,myRing);
ZZ_HYPER_SINGLE_RING(ring2,box);
```

CASE 3:

Array of member class objects:

```
class Obj1 {
    ZZ_EXT_Obj1
public:
```

```
    Obj2 a[20];
    ...
};
class Obj2 {
    ZZ_EXT_Obj2
    ...
};
```

Compare this method with declaring a dynamic array:

```
class Obj1 {
    ZZ_EXT_Obj1
    ...
};
class Obj2 {
    ZZ_EXT_Obj2
    ...
};
ZZ_HYPER_ARRAY(myArray,Obj1,Obj2);
```

When running with C++ but without using *ZZ_INHERIT*, the user must provide a void constructor with *ZZ_INIT()* or *ZZ_OBJECT_ALLOC()* for every *ZZ_EXT_*.. registered class. For examples, see *test11a.c* or *test11b.c*.

8.1.9 ZZansi

This define must be used in the *environ.h* file, if you run with a compiler that adheres to the ANSI standard. This include is required for most C++ compilers.

Except for this one define, do not use the word ZZansi anywhere in your code. If you want to differentiate between ANSI and non-ANSI code, use

#ifdef ZZ_ANSI.

8.1.10 ZZmultiProj

This define must be used in the main header file of a multi-project, where DOL is used hierarchically by several programmers or projects (see an example in directory *orgC/test/multi*). Normally, only one set of internal type tables is kept by DOL, but in the case of a multi-project, type tables are different at each level. In this case, DOL uses tables which are static. For this reason in a multi-project design, larger files with more functions in them are more memory efficient than individual files for each small function. Note that in the multi-project example supplied in the *orgC/test/multi* directory, *-ZZmultiProj* is needed only in the file *orgC/test/multi/main/proj.h.*

8.1.11 ZZnoCheck

This define disables run-time checking. The use of this define is potentially dangerous, but it makes some sense in two situations:

1. When trying to speed-up a well debugged version of the code;
2. when trying to avoid error messages caused by discarding automatically allocated objects.

For example, when leaving the following function

void foo(int i){ Block b; ... }

a C++ compiler automatically deallocates b. If b had been used in some data organizations and had not been explicitely disconnected, this results in an DOL error message.

If the data organization, to which b had been connected, was only temporary, then disconnecting it is a waste of time. If all this is not intentional, however, using *ZZnoCheck* will leave a substantial bug hidden in the code.

In this version, some commands are not sensitive to this option. You can update the files yourself, however, by adding appropriate if-statements inside the macros. For an example, see *macro/delname* or *macro/delslink*.

8.1.12 ZZ_NOLEAK

SUN compilers use a very inefficient (quadratic) deallocator, which is very slow especially if many small objects are being freed. For this reason, the default of OrgC/C++ on SUN is not to free any objects or temporary structures. If you are concerned more about memory than about the time your program needs to run, please specify *#define ZZ_NOLEAK*.

For platforms other than *SUN*, *ZZ_NOLEAK* is the default.

8.1.13 ZZlocal

This define replaces *ZZmain* for a subproject, which is using OrgC/C++ in parallel, but independently from other subprojects. Saving to disk is not permitted on programs running in this mode.

For example, the PAGER organization in *orgC/lib/pager.cc* is coded with OrgC/C++, and though it is a part of the library, it does not interfere with your application code. *#define ZZlocal* forces global declarations to be static.

8.1.14 ZZnoDestr

From Version 3.01, void constructors/destructors containing *ZZ_INIT()* and *ZZ_CHECK()* are generated

automatically. That means that automatically allocated objects are also automatically checked for disconnection before exiting from any function. In some programs, which generate a lot of data without taking care of how to dispose of it on the exit from the program, this full checking generates a lot of error messages, which are irrelevant to what the programmer wants to do. *#define ZZnoDestr* blocks the automatic generation of destructors.

For an example of how to use *ZZnoDestr* together with multiple inheritance, see *test28.c*

8.1.15 ZZbreakLine

This statement is not a define, but a special mark to be used to split long files that must run through *zzprep* into smaller, independently processed segments. Normally, *zzprep*

reads all its input into memory, and processes it as one big string. For large projects, this may be a problem under DOS, where heap data is restricted to under 64k. When *ZZbreakLine* statements are inserted, processing proceeds section by section. This slightly slows processing, but avoids the size limitation.

ZZbreakLine statements must be on a separate line, and start right from the beginning of the line. They also must not split classes or functions. If you want to split the input into many small segments, please use one *ZZbreakLine* statement after each class declaration.

Example: *test25c.c*

8.2 ZZ_EXT_..

For each class which is involved in one or more organizations, you have to use one (and only one) *ZZ_EXT_*.. statement. The ending of this statement must exactly match the class name. When running with C++, use classes, not structures.

Examples:

```
class Plum {
    ZZ_EXT_Plum
    float weight;
public:
    ...
};
class Apple {
    int size;
    ZZ_EXT_Apple
    char colour;
public:
```

```
    ...
};
```

The *ZZ_EXT* statements instruct the class generator on where to insert automatic pointers, that are otherwise transparent. In C++, this statement also inserts the declaration of friends for all related objects. Note that the semicolon must **not** be used after the *ZZ_EXT_* statement. These statements can be anywhere within the structure, but if the *SELF_ID* organization is used on the object, *ZZ_EXT* must be at the beginning of the structure. If this condition is not met, the program detects it at run-time, and prints an error message.

When hyper-objects are used (*ZZ_HYPER_*...), *ZZ_EXT_* should be in the private part of the class.

ZZ_EXT may change *private* into *public*, therefore a new *private* statement must follow after *ZZ_EXT*, if the following variables are to be private.

8.3 ZZ_ORG

These statements are used to declare organizations in C, or in C++ when hyper-objects are not used (Data Manager for C, OrgC, R book), and have a similar purpose to that of *ZZ_HYPER* statements (see below). For example, the statements

ZZ_ORG_SINGLE_RING(eRing,Employee);
ZZ_ORG_TIME_STAMP(Employee);
ZZ_ORG_HASH(eTable,Header,Employee);

are a direct equivalent of

ZZ_HYPER_SINGLE_RING(eRing,Employee);
ZZ_HYPER_TIME_STAMP(Employee);
ZZ_HYPER_HASH(eTable,Header,Employee);

The syntax is exactly the same, but the internal action is totally different. *ZZ_ORG_* is only an instruction for the class generator, and is translated as a comment. *ZZ_HYPER* hides a whole class declaration, including one instance of that class (id).

This is only for your general information, you should not use *ZZ_ORG_* with this tool.

8.4 ZZ_HYPER

ZZ_HYPER statements typically follow object declarations, and declare a hyper-class with one instance of the object. They are equivalent to a database schema, and describe the relations between individual data types. These statements can also be interpreted as instance declarations of abstract organizations.

The table-like form of *ZZ_HYPER* statements and their clarity are important features of DOL.

The syntax of the *ZZ_HYPER* statement is:

ZZ_HYPER_organization(id,type1,type2,..)

where

organization is one of the organizations from the DOL library; *id* is the instance name of this organization, and *type1*, *type2*, ... specify the types to be used.

Each organization has a specific number of types.

Some special hardwired organizations such as *SELF_ID*, *PROPERTY*, or *TIME_STAMP* do not require the *id*, because there can be only one instance of such organizations in any object type.

Otherwise, there is no limit on the number of organizations which are used for any object type. There is also no limit on the number of types used in an organization; in the present library, however, there are no organizations with more than 3 types.

Example 1:

We have four objects: State, Town, Highway, and AirLink. Towns are connected by two networks: Highways and AirLinks. Towns are also grouped by the State to which they belong. All States are linked into a ring, so that they can be sequentially accessed.

ZZ_HYPER_SINGLE_GRAPH(hwy,Town,Highway);
ZZ_HYPER_SINGLE_GRAPH(air,Town,AirLine);
ZZ_HYPER_SINGLE_AGGREGATE(byState,State,Town);
ZZ_HYPER_SINGLE_COLLECT(sRing,Header,State);

Here *hwy* represents a graph, which has towns as nodes and Highways as edges, and *air* represents a graph which has Towns as nodes and Airlinks as edges. *sRing* is a ring of all States, and *byState* is one level of hierarchy, which groups Towns that belong to one State. We added one object, called Header, to encapsulate the entry into the rings of states, *sRing*.

Example 2:

In electrical netlists, such as those used on VLSI chips or printed circuit boards. we deal with Blocks that have Pins. Pins are connected by signal Nets. Both Blocks and Nets must be sequentially accessible, each separately.

ZZ_HYPER_SINGLE_COLLECT(bRing,Header,Block);
ZZ_HYPER_SINGLE_COLLECT(nRing,Header,Net);

ZZ_HYPER_SINGLE_AGGREGATE(byBlock,Block,Pin);
ZZ_HYPER_SINGLE_AGGREGATE(byNet,Net,Pin);

Here, *byBlock* allows access to all the pins of a given block, and *byNet* allows access to all the pins on a Net. *Header* is a dummy object to encapsulate entry points of the two rings.

Example 3:

When dealing with a list of Employee records, we may want to keep a hash table for fast access of the records by name. At the same time, we want to record the time when the objects were created or modified. A hash table must exist as an attribute of some object; object *Header* keeps both the hash table and the entry point for the ring:

```
class Header {
    ZZ_EXT_Header
};
class Employee {
    ZZ_EXT_Employee
        ...
};
ZZ_HYPER_SINGLE_COLLECT(eRing,Header,Employee);<%0>
ZZ_HYPER_TIME_STAMP(Employee);
ZZ_HYPER_HASH(eTable,Header,Employee);
```

Here, *eRing* is the ring of all employees. Each record will have a time stamp. Hash table *eTable* will allow fast access to the Employee records.

IMPORTANT:

Under UNIX, the organization id must not start with a blank character. For example:

ZZ_HYPER_SINGLE_GRAPH(myGraph,Town,Hwy);

results in a compiler error.

8.5 ZZ_LOCAL

In the previous two sections, it was explained that in C, you declare data organizations with *ZZ_ORG_..()*, while in C++ you declare them with *ZZ_HYPER_..()*. Both these declarations have a global character, and even though the interface classes that represent the organization do not contain any data, some users expressed a wish to encapsulate these organizations under some class, to make them completely invisible from the outside.

For this purpose, the class must be designed with some minor differences in how it is constructed, which is exactly what *ZZ_LOCAL_..()* is. *ZZ_LOCAL_..()* has one more parameter than the corresponding *ZZ_ORG_..*or *ZZ_HYPER_..* statements do. The last parameter specifies the class under which the declaration will be hidden (see the example below). The *ZZ_LOCAL_..()* statement must be in the beginning of the class, before *ZZ_EXT_..*

The following example shows how a *RING* can be neatly encapsulated inside a class. This is a much cleaner implementation than you can achieve with standard class libraries. Note that *ZZ_HYPER_UTILITIES()* is always global; functions such as memory management and persistency are difficult if not impossible to make local.

```
// Testing DOL an organization with a local scope
// Here RING myRing is knows only within the scope of class A
#include <<stdio.h>>
#define ZZmain
#include zzincl.h

class A {
    static A *entry;
    int a;
    ZZ_EXT_A
public:
    int val(void){return a;};
    A(){entry=NULL;}
    A(int i);
    A* next(void);
    ZZ_LOCAL_SINGLE_RING(myRing,A,A);
};
ZZ_HYPER_UTILITIES(util);

A* A::next(void){return myRing.fwd(this);}
A::A(int i){a=i; entry=myRing.add(entry,this);}

int main(void){
A aa,*a1,*a2;

a1=new A(1);
a2=new A(2);

a2=a1-.next();
printf("%d %d\n", a1-.val(),a2-.val());

// a1=myRing.fwd(a2); causes compiler error, unknown myRing }
```

```
#include "zzfunc.c"
```

This program is included in the test suite as *test36b.c*.

WARNING:

!!TODO - Check this with Jiri. This style of localizing data organization does not agree well with the basic concept of this entire library, and is not recommended for use. In spite of our effort, some organizations (for example *HASH*) do not work with *ZZ_LOCAL*. Instead of using *ZZ_LOCAL*, we recommend the following strategy, where the organization remains global, but the object that holds its actual implementation is private. All access to the hash table is encapsulated under class Dict. This style naturally fits the design of the library, and the resulting code is clean and simple. (This example was designed in cooperation with Jay Weininger from Reuters.)

Here Dict keeps a hash table of variable length tokens, with only one copy for each string.

```
#include <iostream.h>
#define ZZmain
#include "zzincl.h"
// holder of the hash table|
class Table {
    ZZ_EXT_Table
};
// represents one name entry
class Token {
    ZZ_EXT_Token
public:
    Token(){}
    Token(char *n);
    ~Token();
};
// encapsulates the hash table
// Note the absence of ZZ_EXT_Dict
class Dict {
    Table *tab; // hides the hash table
public:
    Dict(int sz);
    ~Dict();
    void addName(char *n);
    char *getName(char *n);
    void delName(char *n);
};
ZZ_HYPER_HASH(hash,Table,Token);
```

```
ZZ_HYPER_NAME(name,Token);
ZZ_HYPER_UTILITIES(util);
int hash_class::cmp(Token *b1,Token *b2){
    return strcmp(name.fwd(b1),name.fwd(b2)); }
int hash_class::hash(Token *b,int size){
    int ZZhashStr(char *,int);
    return ZZhashStr(name.fwd(b),size);
}
Dict::Dict(int sz){tab=new Table; hash.form(tab,sz);}
Dict::~Dict(){hash.free(tab);}
Token::Token(char *n){
    char* p=util.strAlloc(n); name.add(this,p);}
Token::~Token(){
    char* p=name.del(this); if(p)util.strFree(p);}
void Dict::addName(char *n){
    Token* t=new Token(n);
    Token* p=hash.get(tab,t);
    if(!p)hash.add(tab,t); else delete t;
}
char* Dict::getName(char *n){
    static Token t;
    char *p;
    name.add(&t,n);
    Token* tp=hash.get(tab,&t);
    if(tp)p=name.fwd(tp); else p=NULL;
    (void)name.del(&t);
    return p;
}
void Dict::delName(char *n){
    static Token t;
    name.add(&t,n);
    Token* tp=hash.get(tab,&t);
    if(tp)(void)hash.del(tab,tp);
    (void)name.del(&t);
}
void main(void){
    Dict a(100); char *n;
    a.addName("brown");
    a.addName("black");
    a.addName("brown");
    a.addName("brown");
    a.delName("brown");
    n=a.getName("black"); if(n)cout<<n<<"\n";
```

```
    n=a.getName("brown"); if(n)cout<<n<<"\n";
};
#include "zzfunc.c"
// The program prints only one line: black
```

This example is part of the test suite, see *test36c.c*.

8.6 SYNTAX OF OTHER ZZ CALLS

When using DOL, most of the time, you will need only two additional macros with the prefix *ZZ:* *ZZ_INITIAL()* and *ZZ_CHECK()*. These macros must be inserted into the constructor/destructor for your objects that have *ZZ_EXT* statements. *ZZ_INITIAL()* automatically initializes all internal pointers as disconnected; *ZZ_CHECK()* checks that the object is disconnected from all organizations before it is deallocated.

All other ZZ calls, whether macros or functions, are encapsulated in the hyper-objects and you, as their user, do not have to know about them.

Note that in C++ the C macros for traversing sets, *ZZ_A_TRAVERSE(){ ... }ZZ_A_END* are replaced by iterators, which are automatically created by the *ZZ_HYPER_*.. statements. These iterators do not have the ZZ_ prefix, and behave like normal C++ iterators. For most of the iterators in DOL, the overloaded ++ operator is used to get to the next object in the set.

8.7 VERSION OF THIS LIBRARY

A special global string *ZZorgcVersion*, which appears at the top of the *zzincl.h* file, contains information about the current version of the library. This information allows you to recover the version number for old executables. Use the "strings" program, and search for text "Version".

8.8 *CMP()* AND OTHER AUXILLIARY FUNCTIONS

Both the documentation and the test programs frequently use auxilliary free floating functions, such as the compare function needed for sorting (equivalent to the *cmp()* function needed for qsort). Similar functions are needed when you want to sort any of the organizations in this library, or when you want to use the binary heap or the hash table.

A much better coding style is not to use free floating functions, but to make these functions static and assigned to the class of objects that are being compared. Both styles are supported by this library. For examples of the better style - see the example in Chap.7, or *test0r.c*.

The advantages of not using virtual *cmp()* functions as it is common in some other class libraries are that:

- If you don't sort your objects, you do not have to provide dummy *cmp()* functions that only clutter the code.
- If your classes don't have virtual functions, you avoid the unnecessary overhead (4 bytes for each subclass - a significant amount of memory for composite objects).

Note that the *cmp()* and *hash()* functions for hash tables are assigned to the organization itself (see Chap.14.5), and therefore must be coded as:

inline int _class::cmp(oType*,oType*);

inline int _class::hash(oType*,int);

« Chapter 7: Examples

Chapter 9: Single User Mode »

9. SINGLE USER MODE

This chapter describes how to run OrgC++ in the single user mode. For a more complicated case where several programmers use OrgC++ simultaneously, see Chap.18.

The general syntax for invoking the class generator is:

orgCpath/zzprep prog.cpp <incl.h> <master> <func.c> <comb.h>

where:

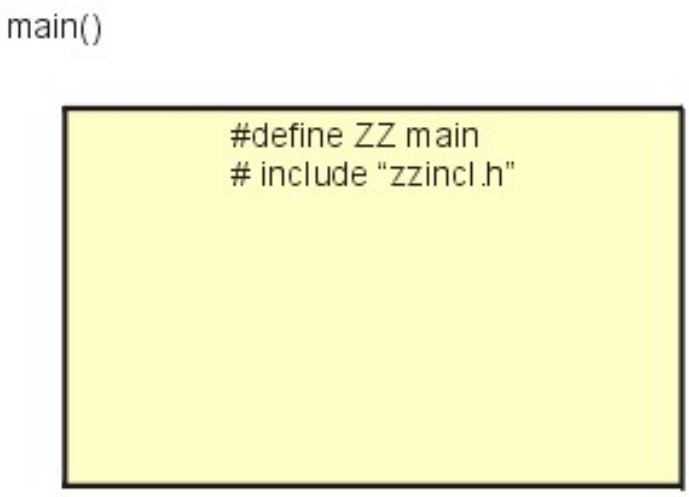
orgCpath is the path to the orgC directory,
prog.cpp is either the total source, or the file containing object declarations with all *ZZ_EXT* and *ZZ_HYPER* statements,
incl.h is the result of preprocessing (default=*zzincl.h*),
master is the master table of the chosen library (default=*zzmaster*).
func.c is the second file produced by the class generator (default=*zzfunc.c*).
comb.h is the reduced macro file (default=*zzcomb.h*).

WARNING: The full path to the orgC directory must be given in the invocation of *zzprep*. It is not sufficient to set a path to the orgC directory, and then simply call *zzprep*. The *zzprep* program needs the full path to derive the locations of some additional files it needs for operation, in particular the file *orgC/macro/zzmaster*. If you get an error message about the program not being able to read this file, most likely the problem is in how you call *zzprep*.

9.1 ALL CODE IN ONE FILE

Under UNIX, we assume both *mySource.c* and *mySource.cc* will be interpreted as C++ source files, depending on the content of the file. Under DOS, it is automatically assumed that *mySource.cpp* is a C++ file, and that *mySource.c* is a C source file.

Start your program like this:




```
Run the whole source through the class generator:
    orgC/zzprep mySource.c ... under UNIX
    orgC\zzprep mySource.cpp ... under DOS
Compile (under UNIX):
    CC mySource.c orgC/lib/zzlib.a
Compile (DOS, TurboC++):
    tcc -mm -Lc:\tc\lib -Ic:\tc\include mySource.cpp orgC\lib\zzlib.lib
Compile (DOS, BorlandC++):
    bcc -mm -Vt -Lc:\tc\lib -Ic:\tc\include mySource.cpp orgC\lib\zzlib.lib
Compile (DOS, ZortechC++):
    ztc -mM mySource.cpp /link orgC\lib\zmlib.lib
```

EXAMPLE: See *test0a.c* in your *orgC/test* directory.

The preprocessor (*zzprep*) skips over sections of code commented out by either /*...*/ or by //... However, the preprocessor does not react to *#ifdef* statements. For example, in the following situation

#ifdef SKIP

....some code

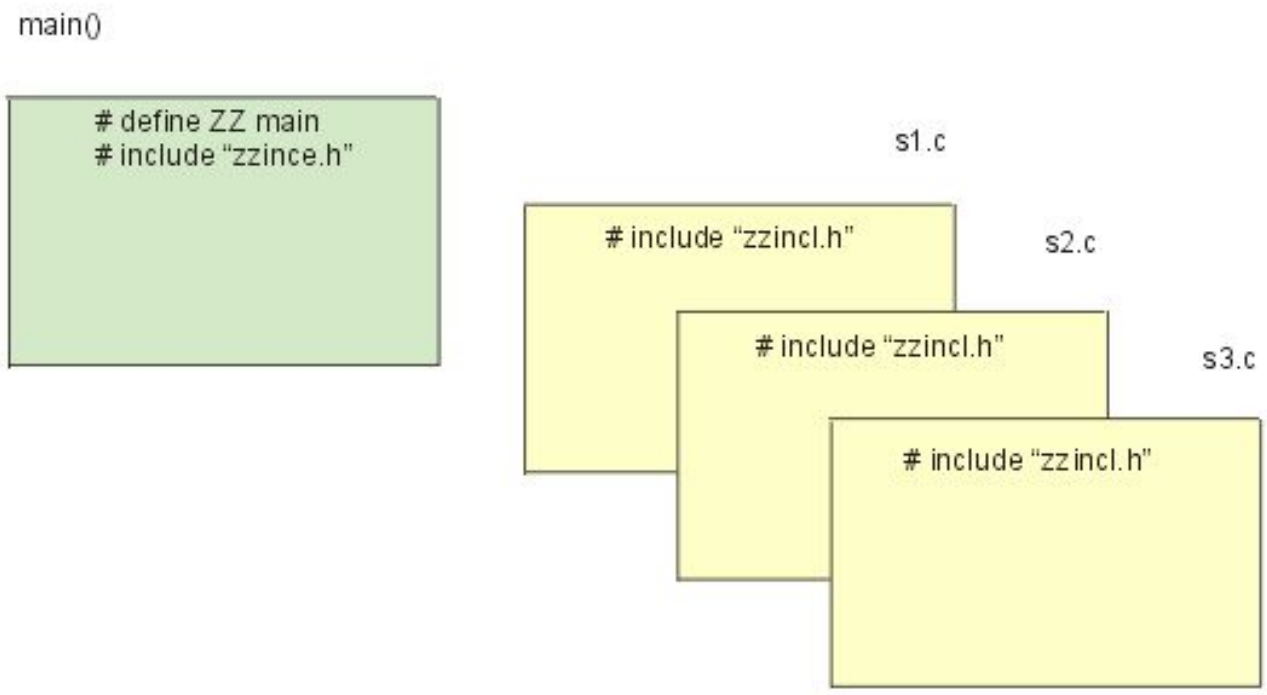
#endif

the preprocessor will run through all the code, regardless whether SKIP is defined or not.

9.2 LARGER PROGRAMS

The class generator does not have to read all your source, it needs only those sections of your code, and those include files that use ZZ-prefix commands. These commands are *ZZ_EXT_..*, *ZZ_HYPER_..*, and possibly *ZZ_CHECK()*. In OrgC++, such statements are used in class declarations only, and should therefore be concentrated in a few header files.

Regardless of how many source files you use, you proceed in this manner, adding *cat* (or *type* under DOS) statements as required:



Combine all your include files into one large file, and run this file through the class generator.

The source files must start like this:!!TODO - Ask Jiri if something is missing.

Then you compile and link your program as usual. Note that instead of including *zzfunc.c* at the end of the *main* program, you can compile it separately.

EXAMPLE: *test7a.c*, *test7x.c*, and *test7.h* in your *orgC/test* directory.

This method has two advantages: (a) the class generator runs faster, because the combined include file is always short compared to the overall source; (b) you don't have to re-run the class generator during debugging, unless you also touch one of your include files. This is a condition which may also be used by the *Makefile* facility.

9.3 SPLITTING LARGE FILES

Some compilers, especially under DOS, run out of space when applied to a large file which contains many functions. The solution to this problem is to split the file into several files, each containing fewer functions and fewer lines of code.

You can encounter this problem in two situations:

1. The input file for A *zzprep* is too large, and *zzprep* runs out of memory. In this case, use *ZZbreakLine* statements to divide the input file into segments that are processed individually. For more details, see Chap.8.1.15.
2. *zzprep* generates the file *zzfunc.c* which is too large, and the compiler runs out of memory.

File *zzfunc.c* is normally out of your control, and even if you edit it manually and split it into several sections, you would have to do the same work again any time *zzprep* is invoked.

The self-standing program *zzsplit* allows you to split *zzfunc.c* into several sections of approximately the same size. All the sections are placed into the same file (default name *zzfun.c*) with individual sections separated by *#ifdef* statements:

```
#ifdef SECTION1
...
#endif
#ifdef SECTION2
...
#endif
#ifdef SECTION3
```

... and so on

If you want to compile the first section, create a simple file which contains the following two lines, and compile is separately:

```
#define SECTION1
#include "zzfun.c"
```

The general syntax for the *zzsplit program* is:

```
    zzsplit zzfunc.c zzfun.c inp
```

where

zzfunc.c is the original (normal) *zzfunc.c* file;
    default: zzfunc.c
zzfun.c is the new file with special *ifdefs*;
    default: zzfun.c
inp is the input file containing one integer, number of required sections;
    default: stdin

EXAMPLE:

If you want to split *zzfunc.c* into 4 sections, and use the defaults for *zzfunc.c* and *zzfun.c*, create a simple file inp which contains just one number, 4. Then call

zzsplit inp

This technique is used in test18f, see *orgc/test/cregr* or one of the other regression files.

10. ORG C++ LIBRARY

OrgC++ includes a library of organizations. The library is open: you can add new organizations and functions.

The library has two parts:

1. The *orgC/macro* directory contains the source code for all macros and parametric functions. (Examples: *add1ton* and *fadd1ton.*)
2. The *orgC/lib* directory contains the source and libraries (**.a* files under UNIX, **.lib* file under DOS) for more complicated features, such as hash tables or saving to disk.

Note that each file in the *macro* directory contains both complete documentation (including an example) and the executable code. The file *macro/zzmaster* contains the master table for the whole library. It contains the following sections:

- list of currently available organizations;
- list of pointers used by individual organizations;
- list of functions/macros in the library;
- list of support include files.

For more details about this file, see Chap.16.

Even the hyper-object definitions (*ZZ_HYPER_.. calls*) must be registered in the file *zzmaster*. The class generator needs this information on the internal setup of each hyper-object.

When you link a program written with OrgC++, you have to link it to the appropriate library:

UNIX:

orgC/lib/zzclib.a

DOS, TurboC++:

orgC\lib\cmlib.lib for the medium memory model
orgC\lib\cllib.lib for the large memory model

DOS, ZortechC++:

orgC\lib\zmlib.lib for the medium memory model
orgC\lib\zllib.lib for the large memory model

For information on how to recompile the library see Chap.2 (Installation).

If you have been using OrgC with regular C, and you are now transferring to OrgC++, look at Appendix A. It contains a table of all functions and their uses in C and C++.

See Chap.12 (on-line help) on how to query the library documentation.

When calling the class generator you may, for the third parameter, specify the *master* file you want to use:

orgCpath/zzprep source.c ZZinclude.h zzmaster ZZfunc.c

This permits you:

1. to select only a subset of organizations and simplify the preprocessing;
2. to merge the standard OrgC++ library with one or more user libraries.

For more details on how to select/merge libraries, see Chap.17.

« Chapter 9: Single User Mode

Chapter 11: Available Organizations »

11. AVAILABLE ORGANIZATIONS

This chapter describes the data organizations and hardwired features currently available from the library.

Only a conceptual introduction to each organization is presented. For programming details, including examples, use one of the following sources:

- on-line help (Chap.12);
- browse files in the *orgC/macro* directory;
- reference guide, *orgC/docum/ZZrefer* (Chap.13);
- the *orgC/test* directory, use *grep ... *.c*.

Although the library already contains all the essential organizations, it is still growing. We provide our active users with regular updates.

All OrgC++ organizations are based on a ring-type arrangement, not on a NULL-ending list. The two arrangements are similar in access and performance, but a RING has several advantages: head and tail are represented by the same pointer, handling is more uniform, and a NULL pointer cannot occur within a valid organization. The last property is the key to run-time protection against dangling pointers in OrgC++.

With the exception of the *GENERAL_LINK*, OrgC++ structures are strongly typed. This has the advantage of the code being thoroughly checked by the compiler, but requires special handling of heterogenous objects. For example, if we declare

```
ZZ_HYPER_SINGLE_RING(aRing,Apple);
ZZ_HYPER_SINGLE_RING(oRing,Orange);
```

aRing may contain only *Apples* but no *Oranges*, and *oRing* may contain only Oranges but no *Apples*. Neither of the two rings may contain *Plums* or *Pears*.

Heterogeneous situations can be handled easily either by using a virtual function - see the example in Chap. 7.3- or by combining *GENERAL_LINK* with other OrgC++ organizations. For example, in addition to *Apples* and *Oranges*, we can declare a general object called *Fruit*. Then

```
ZZ_HYPER_SINGLE_RING(fRing,Fruit);
ZZ_HYPER_GENERAL_LINK(fLink,Fruit);
```

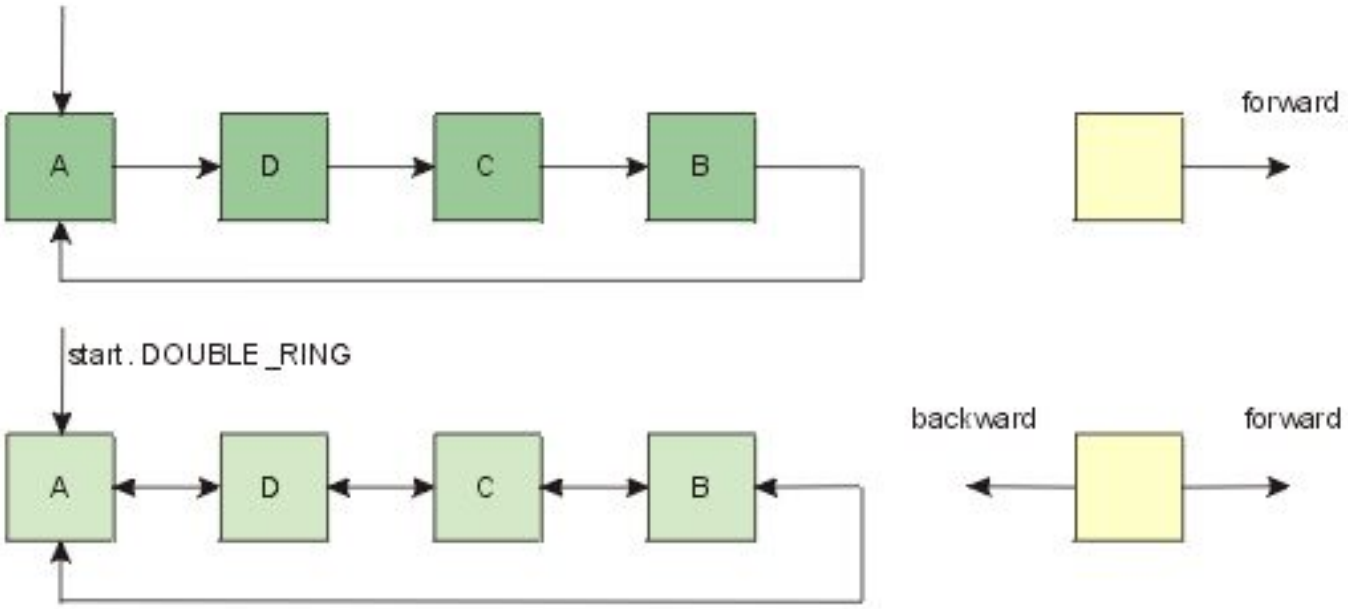
describes a RING of Fruits, that may be anything, including *Pears* or *Plums*. *GENERAL_LINK* provides the link between the general object (*Fruit*) and the particular object (*Apple*,*Orange*, *Plum*, ...).

Most organizations automatically provide an iterator, which lets you to walk through the entire data set. Singly linked organizations let you walk only head to tail, using either function *fwd()* (short for 'forward') or iterator operator ++. For convenience, we recommend you to use macro *ITERATE(iterator,objPtr){...}* which hides a for loop using operator ++. For examples, see *orgc/test/test50.c*.

Doubly linked organizations allow you to traverse the data also in the reverse direction, using either function *bwd()* (short for 'backward') or iterator operator--!!TODO - Check with Jiri. Again, we recommend you using convenient macro *RETRACE*, which is just like *ITERATE*, except that it traverses the data tail to head.

Macros *ITERATE* and *RETRACE* hide normal for(..) loops, and therefore can include *continue* statements. These loops can also be nested at any number of levels. Note however that operators ++ and -- are designed only for straight traversal of the whole set. For example, you cannot mix the use of ++ and -- within one loop. If you need to perform some complex traversal of the data, going back and forth over it, use functions *fwd()* and *bwd()* they have no limitations.

11.1 RING



Purpose:

List - *RING* can serve the same purpose as a NULL-ending list;
Queue - objects can be retrieved in the same/reverse order as they are loaded.

IMPORTANT: This organization represents a ring without the encapsulated entry point, which must be managed externally. If you need a fully encapsulated ring, look at *COLLECTION* (Chap.11.2).

SINGLE_ and DOUBLE_RING:

Both *RINGS* behave identically, except that the *DELETE* operation is much faster for the *DOUBLE_RING*. In applications where *DELETE* is not used frequently, *SINGLE_RING* is a better choice (it is faster, and needs less memory).

Sorting:

The *sort()* function sorts a given ring. You provide your own compare function, as you do with *qsort()*. See the example in *test 15a.c*. Note that *sort()* which is based on the merge algorithm, is generally faster than *qsort()*.

Adding and deleting objects:

In our library, a ring (or circular list) is a structure existing on a set of objects, without any start/tail pointer encapsulated in a special class. If you want a ring with an encapsulated entry point, use *COLLECTION* or *AGGREGATE*.

Since the entry to the ring is not encapsulated, you have to keep it externally yourself, otherwise the ring

would be there, but you would not know how to get to it. The entry is also important if you are concerned about the order of the objects in the ring. The entry to the ring will be returned as the last element when traversing the ring.

Before you start to use any ring, set *entry=NULL*. Use *add()* to add an object after *start*, *ins()* to insert it before *entry* (you can only do that for a *DOUBLE_RING*), and *del()* to delete an object.

Order of objects:

If you represent a *RING* as a sequence of objects, starting from the *entry* object, then the *RING* behaves in the following manner:

empty RING	adding A gives	A
A	adding B gives	AB
AB	adding C gives	ACB
ACB	adding D gives	ADCB
ADCB	deleting C gives	ADB
ADB	deleting A gives	BD

If you want to return objects in the same order as they were added to the ring, two methods can be used: Either use a *DOUBLE_RING* and the iterator instead of the ++ iterator (see below on the interators), or reset the *entry* after each object is added. The first method works for a *DOUBLE* ring only, the second method works for both *SINGLE* and *DOUBLE* rings. Here is an example of the second method (resetting the entry point):

```
entry=NULL; /* initialize start before using the ring*/
...
entry=myRing.add(entry,p1);
entry=p1;
...
entry=myRing.add(entry,p2);
entry=p2;
...
```

This method will store objects in the order in which they were loaded, because the *entry* represents the 'tail' of the ring.

empty RING	adding A gives	A
A	adding B gives	BA

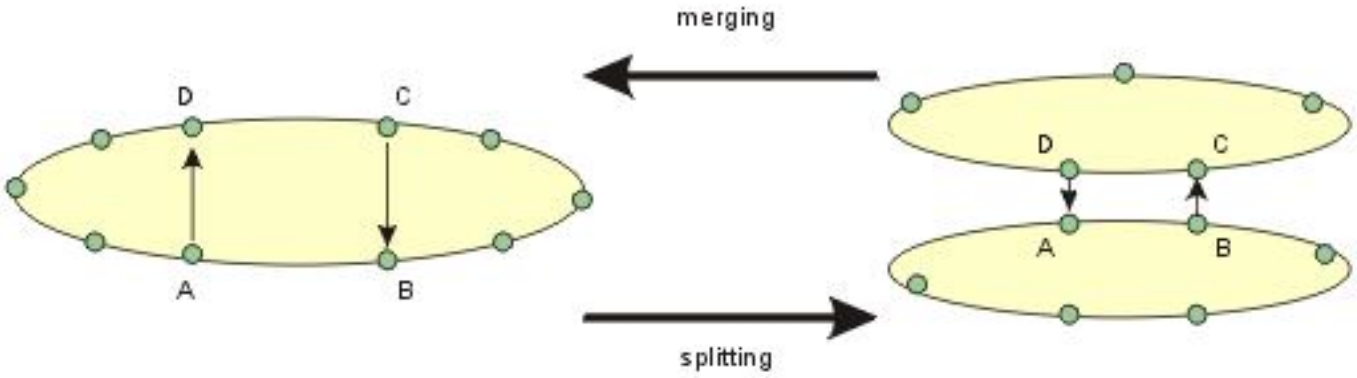
AB	adding C gives	CAB
ACB	adding D gives	DABC

When running this sequence through the *iterator* you get: A,B,C,D.

Merging or Splitting Rings

It is interesting (see the diagram on the following page) that the same operation with pointers provides merge/split operations, depending on whether the two given points are in the same ring, or in two different rings.

Algorithm: For given points A and C, disconnect pointers to B (which is next to A) and to D (which is next to C). Then connect A to D and C to B.



If A and C were originally on the same ring, the operation results in splitting the ring; A is on one ring, and C is on the other ring. If A and C were originally on different rings, the operation merges the two rings into one. One command, *merge()*, serves both operations. If you want to extract a single object and make it a new ring, do this:

```
ZZ_HYPER_SINGLE_RING(myRing,Obj);
...
Obj *p1,*p2;
p2=myRing.fwd(p1);
p1=myRing.merge(p1,p2);
// now p2 forms a single object ring,
// remaining objects are on the p1 ring
```

Moving around the *RING*:

Use an iterator to move around a *RING*:

```
myRing_iterator it(p1);
```

while(p2= ++it){
...
}

For example, for ring ADCB, this loop returns D,C,B,A.

fwd() returns the next object in the chain (right neighbour).
bwd(), which is defined only for the organization *DOUBLE_RING*, returns the previous object, and allows traversal of objects in reverse order. *del()* can be used while traversing the set with the *iterator*.

If you call *bwd()* for a *SINGLE_RING*, the compiler will detect an error.

Syntax:

The syntax is the same for single/double rings:

ZZ_HYPER_SINGLE_RING(id,TYPE);
ZZ_HYPER_DOUBLE_RING(id,TYPE);

declare singly/doubly-linked rings,

TYPE id.add(TYPE *entry,TYPE *newObj);*

Adds new object *newObj* to the ring with entry point *entry* and returns a new *entry*. For an empty ring, set *entry=NULL* before this call.

TYPE id.del(TYPE *entry,TYPE *remObj);*

Deletes *remObj* from the ring, and returns the new tail of the ring. If *remObj* is the tail before the call, the new tail will be different. If *remObj* is the last object in the ring, the function returns NULL.

TYPE id.sort(int (*cmpFun)(const void*,const void*), TYPE *entry*);

Sorts a ring, using *cmpFun()* to compare objects. In most cases, *entry* will change after this call.

TYPE id.merge(TYPE *obj1,TYPE *obj2);*

If the two objects are on the same ring, this splits the ring into two. If the objects are on different rings, it merges the rings. Always returns *obj1*.

TYPE id.fwd(TYPE *obj);*

For a given object, it returns the next object on the ring. If the object is not connected to any ring, it returns *next=NULL*

```
TYPE *entry,*obj
id_iterator it(entry);
while(obj= ++it){ ... };
```

This traverses the ring with entry point *entry*.

```
it.start(entry); //restarts the same loop
```

Commands available only for the *DOUBLE_RING*:

```
TYPE* id.bwd(TYPE *obj);
```

For a given object, the function returns the previous object in the ring. If the object is not in any ring, it returns *NULL*.

```
void id.ins(TYPE *obj,TYPE *newObj);
```

Inserts *newObj* before *obj*.

```
TYPE *entry,*obj;
id_iterator it(entry);
while(obj=it---){ ... };
```

This traverses the ring in the opposite direction.

```
it.start(entry); //restarts the same loop
```

Examples:

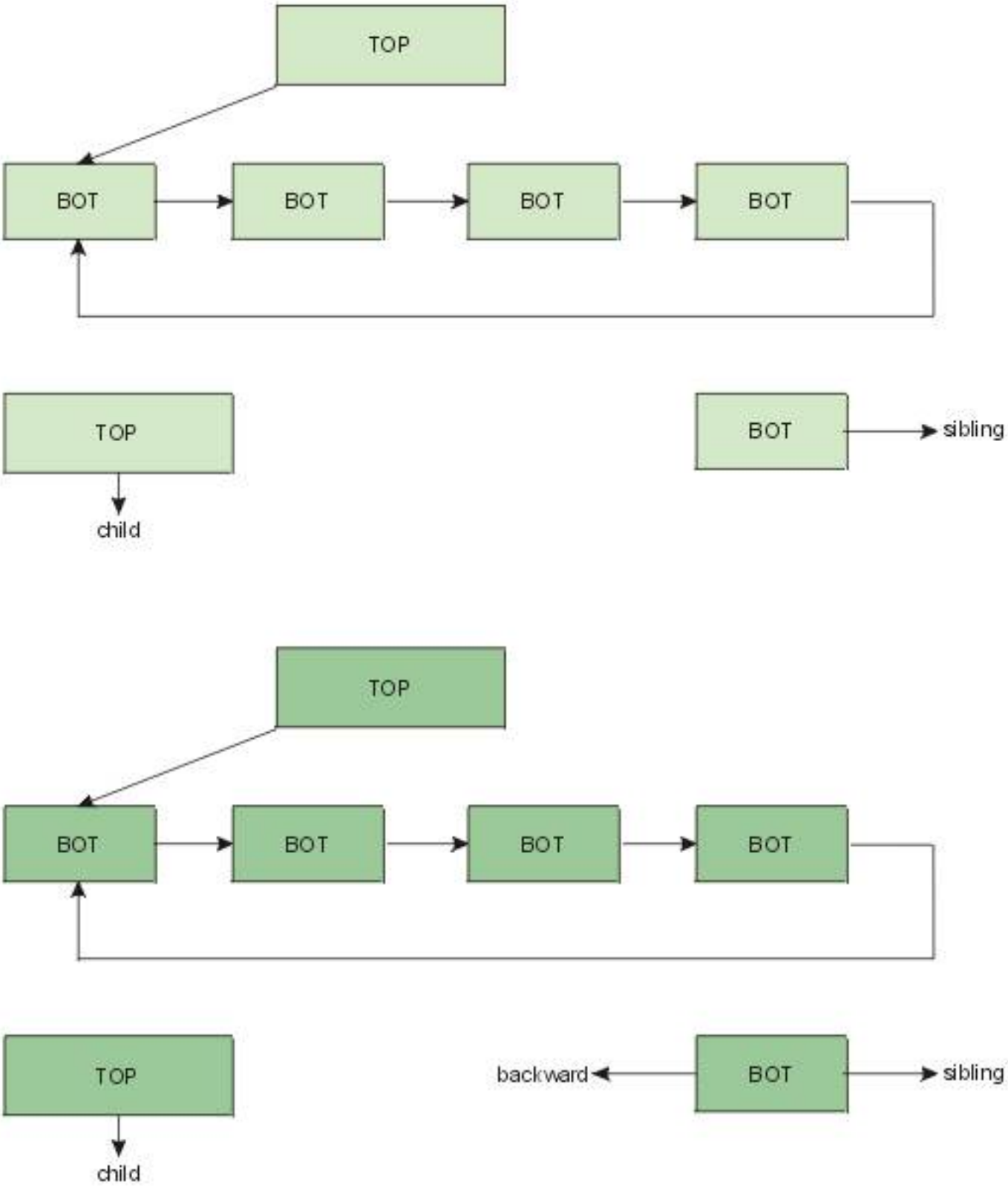
1. List of cell instances and list of signal nets on a VLSI chip (*test2.c*).
2. Employee records sorted into two rings: one by employee name and one by salary (*test15a*).
3. Loading Blocks and Nets into netlist without reversing the order (*test0a.c*).

Warning: The ZORTECH C++ version of OrgC++ does not contain the *sort()* function. Instead, the macro *ZZ_SORT()* must be used, as is used in plain OrgC.

Chapter 10: Org C++ Library

Next Section: 11.2 COLLECTION

11.2 COLLECTION



Purpose:

An ordered or unordered *COLLECTION* occurs when one object type (TOP) contains a whole set of objects of another type (BOT).

This organization can also be used as a *RING* with an encapsulated entry point. If the ring is singly linked it is a single collection(*SINGLE_COLLECT*), if it is doubly linked it is a double collection (*DOUBLE_COLLECT*).

Note the difference between a *COLLECTION* and a *TREE*. A *TREE* is composed of objects that all have the same type; a *COLLECTION* works with two different object types. Also, a *COLLECTION* requires only about 50% of the memory required by a tree because the child and parent pointers on the bottom objects are missing.

You can also look at the *COLLECTION* as being a *TRIANGLE* (see Chap.11.3) where the parent pointer is missing, or as a combination of singly-linked *RINGS* (Chap.11.1) with a single *LINK* (Chap. 11.6).

ADD and DELETE:

These commands add/delete objects to/from the bottom level. In order to make a *COLLECTION* empty, delete all its bottom objects. *del()* can be used while traversing the set with the *iterator*.

Order of objects:

The objects at the bottom level are handled as a ring. If objects ABCD have been added to a *COLLECTION*, the *iterator* returns them in the reverse order: DCBA. The *child* pointer of the top object works as the *start* pointer of the *RING*. *set()* allows you to reset this pointer. Therefore, for example, the following sequence results in the bottom objects being ordered in the same sequence as they were loaded:

```
ZZ_HYPER_SINGLE_COLLECT(myCol,ParType,ChiType);
ParType *p; ChiType *c1,*c2;
...
myCol.add(p,c1);
myCol.set(p,c1);
...
myCol.add(p,c2);
myCol.set(p,c2);
...
```

sort() sorts all the children using a user supplied compare function similar to the one needed for *qsort()*.

Moving around the *COLLECTION*:

The iterator traverses the bottom elements of the *COLLECTION*.

myCol_iterator it(p);
while(c1= ++it){
...
}
fwd() returns the next object in the bottom ring;
child() returns the starting bottom object.

Merging or Splitting Collections

Because all the children of a collection form a ring, collections can be merged or split just like a ring (see the illustration in Chap.11.1).

The merge command must be given two children and one parent object. If the two children are in the same collection, the operation results in splitting the collection in two. If the two children are in different collections, the operation merges them into one.

id.merge(c,c,par) results in no action. In order to extract a single object into a new *COLLECTION*, proceed as for the *RING*.

In the case of merging, the given parent must be the parent of the second child. In the case of splitting, an empty parent must be given to receive the second part of the set.

```
ZZ_HYPER_SINGLE_COLLECT(myCol,ParType,ChiType);
ParType *p1,*p2;
ChiType *c1,*c2 // given two children
    ....
// p1 and p2 are parents of two collections
// c1 is child of p1, c2 is child of p2
if(p1==p2){ // case of splitting
    p2=new ParType;
    c1=myCol.merge(c1,c2,p2);
    // p1 and p2 now hold the two collections
}
else { // case of merging, p2 must be parent of c2
    c1=myCol.merge(c1,c2,p2);
    // p1 contains the result, p2 is empty
}
```

Syntax:

ZZ_HYPER_SINGLE_COLLECT(id,TOP,BOT);	Declares a singly-linked collection.
*void id.add(TOP *parent,BOT *newChild);*	Adds *newChild* to the given parent.

*void id.del(TOP *parent,BOT *oldChild);*	Deletes *oldChild* from this *parent*.
BOT id.fwd(BOT *child);*	Returns next child (fwd=FORWARD) under the same parent.
BOT id.child(TOP *parent);*	Returns the first *child* under this *parent*.
*void id.set(TOP *parent,BOT *newFirstChild);*	Sets already existing *child* as the new first *child* of this *parent*

Warning: Using an improper parent will cause an un-detectable error.

*void id.merge(BOT *child1,BOT *child2,TOP *parent);*	If the two children are from the same collection, this splits it into two parts, and *parent* will be the second collection. If the two children are from different collections, the collections will merge.
*void id.switchParents(TOP *parent1,TOP *parent2);*	Exchanges parents of two collections. When one collection is empty, this effectively moves a collection under a new *parent*.
*void id.sort(int (*sortFun)(const void*,const void*),TOP *parent);*	Sorts the elements of the collection, using *sortFun()* to compare the objects.
*TOP *parent; BOT *obj;* *id_iterator it(parent);* *while(obj= ++it){ ... }*	This loop traverses all the elements of the collection. The value of *obj* should not be changed from within the loop. Upon exit from the loop, *obj=NULL*.
it.start(parent);	Restarts the same loop.

Available only for the *DOUBLE_COLLECT*:

ZZ_HYPER_DOUBLE_COLLECT(id,TOP,BOT);	Declares a doubly-linked collection.
BOT id.bwd(BOT *child);*	For a given *child* returns the previous (*bwd=BACKWARD*) *child* in the ring that forms the collection.

*void id.ins(BOT *child,BOT *newChild);*	Inserts the new *child* before the given *child.*

*TOP *parent; BOT *obj;*
id_iterator it(parent);
while(obj=it--){ ... }
	This loop is just like the one above, except that it traverses the data in reverse order.

it.start(parent);	Restarts the same loop

Most new C++ compilers for DOS issue warnings when using the *while()* statement in this situation. You can use the *iterator* in a different form which is not as easy to read, but it does not generate any warnings:

id_iterator it(parent);
for(child= ++it; child; child= ++it){ ... };

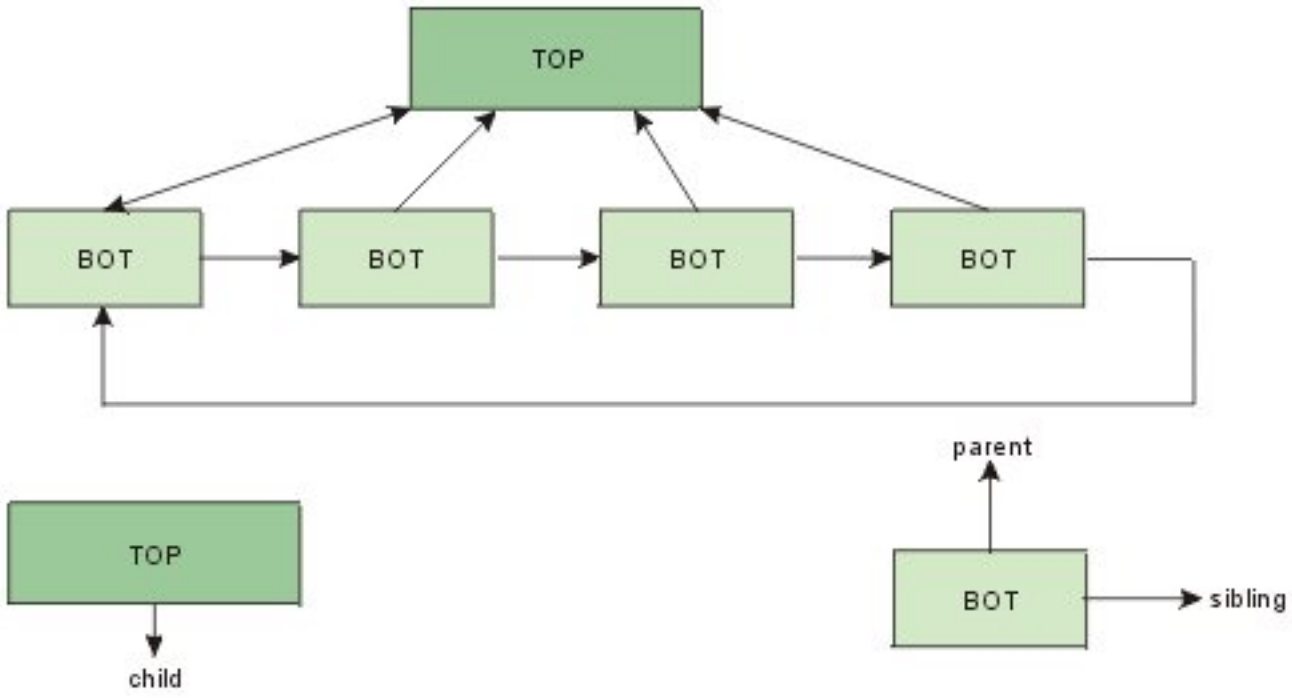
Examples:

Netlist in electrical circuits, storing pins by block and by signal net (*test0i.c*).

Previous Section 11.1 RING

Next Section: 11.3 AGGREGATION

11.3 AGGREGATION (TRIANGLE)



Purpose:

A triangle represents one level of a general hierarchy, with one object type at the top (*TOP*), and another object type at the bottom (*BOT*).

Note the difference between a *TRIANGLE* and a *TREE*. A *TREE* is composed of objects that all have the same type; a *TRIANGLE* works with two different object types. Also, a *TRIANGLE* requires about 30% less memory than a *TREE* because the *child* pointer on the bottom objects is absent.

ADD and DELETE:

These commands add/delete objects to/from the bottom level. In order to make a *TRIANGLE* empty, delete all of its bottom objects. *delete()* can be used while traversing the set with the *iterator*.

Order of objects:

The objects on the bottom level are handled as a ring. If objects ABCD have been added to a *TRIANGLE*, the *iterator* returns them in reverse order: DCBA. The *child* pointer of the top object works as the *start* pointer of the *RING*. Function *set()* allows you to reset this pointer. Therefore, for example, the following sequence results in the bottom objects being ordered in the same sequence as they were loaded:

```
...
id.add(par,obj1);
id.set(obj1);
```

```
...
id.add(par,obj2);
id.set(obj2);
...
```

The function *sort()* sorts all children using a user supplied compare function similar to the one needed for *qsort()*.

Moving around the TRIANGLE:

The *iterator* traverses the bottom elements of the *TRIANGLE*.

```
id_iterator it(parent);
while(obj= ++it){
...
};
```

fwd() returns the next object in the bottom ring;
par() returns the parent object for a given bottom object;
child() returns the starting child object.

Merging or Splitting Triangles

Because all the children of a triangle form a ring, triangles can be merged or split just like a ring (see the illustration in Chap.11.1).

The merge command must be given two children and one parent object. If the two children are in the same triangle, the operation results in splitting the triangle in two. If the two children are in different triangles, the operation merges them into one.

id.merge(c,c,par) results in no action. In order to extract a single object into a new *TRIANGLE*, proceed as shown for the *RING*.

When merging, the given *parent* must be the parent of the second *child*. When splitting, an empty parent must be given to receive the second part of the set.

```
ZZ_HYPER_SINGLE_TRIANGLE (myTriangle,parType,chiType);
parType *p1,*p2;
chiType *c1,*c2;
    ....
p1=myTriangle.par(c1);
p2=myTriangle.par(c2);
if(p1==p2){ // case of splitting
    p2=new parType;
```

```
        myTriangle.merge(c1,c2,p2);
        // p1 and p2 now hold the two triangles
    }
    else { // case of merging
        myTriangle.merge(c1,c2,p2);
        // p1 contains the result, p2 is empty
    }
```

Syntax:

ZZ_HYPER_SINGLE_TRIANGLE(id,TOP,BOT); *ZZ_HYPER_SINGLE_AGGREGATE(id,TOP,BOT);*	Both statements declare identical organizations (both names are acceptable).
*void id.add(TOP *parent,BOT *newChild);*	Adds new *child* under the given *parent*.
*void id.del(BOT *childToRemove);*	Removes the given *child* from the aggregation.
BOT id.fwd(BOT *child);*	Returns the next *child* under the same aggregation (*fwd=FORWARD*)
TOP id.par(BOT *obj);*	Returns the parent of the given object.
BOT id.child(TOP *obj);*	Returns the first child of the given object.
void id.set(newFirstChild);	Sets the given object as the first *child* under the same aggregation.
*void id.merge(BOT *child1,BOT *child2,TOP *parent);*	If the two children are in the same aggregation, the aggregation will split, and *parent* will be the parent of the new section, which will contain *child2*. If the two children are in different aggregations, the two aggregations will merge, and the *parent* of *child1* will be its *parent*.

*void id.switchParents(TOP *parent1,TOP *parent2);*	Exchanges parents of two aggregates. If one aggregate is empty, it effectively moves an aggregate under a new *parent*.
*void id.sort(int (*sortFun)(const void*,const void*),TOP *parent);*	Will sort children under the given *parent*, using the given function to compare them.
*TOP *parent; BOT *obj;* *id_iterator it(parent); while(obj= ++it){ ... };*	Will traverse all children of the given parent. *obj* must not be modified from within the loop, but objects can be deleted using *id.del()*. You can break from the loop as usual.

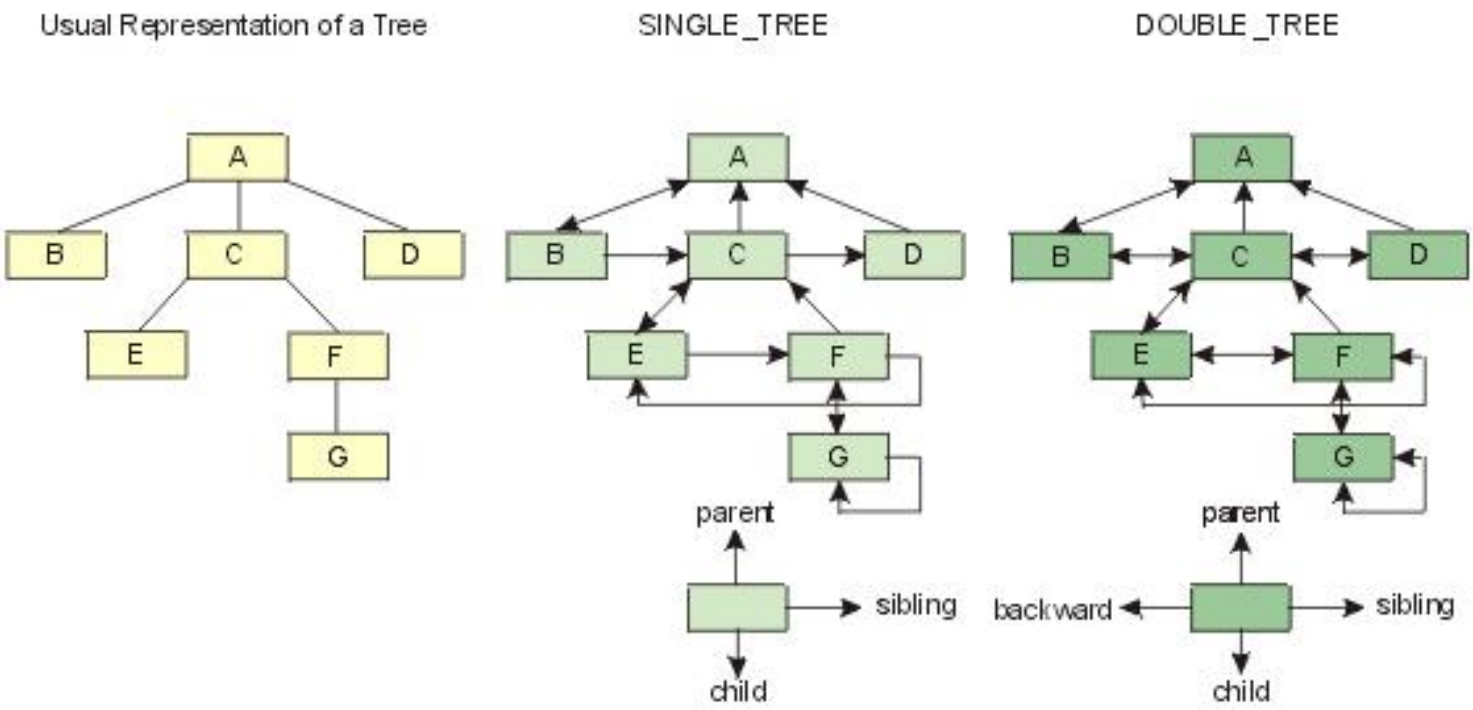
Comment: *id.sort()* does not work for the ZORTECH compiler.

Examples:

1. Netlist in electrical circuits, storing pins by block and by signal net (*test0a.c*, *test2.c*).
2. Storing a list of towns under the state to which they belong (*test 14a.c*).
3. Loading Pins by Blocks and by Nets into netlist, without reversing their order (*test0a.c*).

Previous Section 11.2 COLLECTION Next Section 11.4 TREE

11.4 TREE



Purpose:

General tree representation is useful in representing multiple level hierarchies, and is also important for many algorithms based on the divide-and-conquer approach. All the objects in the tree must be of the same type.

SINGLE_ and DOUBLE_TREE:

Both *TREES* behave identically, except that the *DELETE* operation is faster for the *DOUBLE_TREE*. In applications where *DELETE* is not frequently used, *SINGLE_TREE* is the better choice (it is faster, and needs less memory).

Order of objects:

All the children of any node form a *RING*. The *child* pointer from the *parent* represents the *start* pointer of the *RING*. Repeated use of *add()* loads the nodes in reverse order. *set()* can be used after each *add()* - refer to *TRIANGLE* - if the order is not to be reversed.

Adding new nodes:

add()	adds a new child;
set()	resets the beginning of the given subtree;

app()	appends a new sibling to the right of a given node.
ins()	(for DOUBLE_TREE only) inserts a new sibling to the left of a given node.

Deleting nodes:

del() disconnects the given node from its *parent*, but if there are any subtrees on it, it leaves them on the node. If you call *del()* on the root of a tree, nothing happens. *del()* can be used while traversing the set with the *iterator*. To disconnect a whole tree, move recursively bottom up like this:

```
ZZ_HYPER_SINGLE_TREE(myTree,Node);
void deleteTree(Node *root){
    Node *ep;
    Node* child=myTree.child(root);
    if(!child)myTree.del(root);
    else {
        myTree_iterator it(root);
        while(ep= ++it)deleteTree(ep);
    }
}
```

Moving around the TREE:

The iterator traverses the subnodes of a given node.

```
id_iterator it(parent);
while(obj= ++it){
...
};
```

This can be used recursively to traverse the whole tree (breadth first, depth first, whatever you prefer). See, for example, function *prt3()* in *test12a.c*.

par(), child(), fwd() and *bwd()* (for *DOUBLE_TREES* only) allow you to move around the tree. *fwd()* moves to the right, *bwd()* moves to the left. Remember that because each set of siblings is a *RING*, repeated calls to *bwd()* will lead to an infinite loop, unless proper termination is arranged. The *iterator* provides automatic termination when reaching the starting point.

Syntax:

ZZ_HYPER_SINGLE_TREE(id,TYPE); *ZZ_HYPER_DOUBLE_TREE(id,TYPE);*	These statements declare a tree, with children of each node linked into a singly/doubly linked ring.

*void id.add(TYPE *parent,TYPE *newChild);*	Adds a new *child* under the given *parent*.
*void id.app(TYPE *obj,TYPE *newObj);*	Appends a new object after the given object (under the same *parent* node).
*void id.del(TYPE *objToRemove);*	Disconnects the given object from its *parent* and siblings, but it leaves its children and possibly a whole subtree.
TYPE id.fwd(type *obj);*	Returns the next sibling under the same parent (*fwd=FORWARD*).
TYPE id.par(TYPE *obj);*	Returns the *parent* of the given object.
TYPE id.child(TYPE *obj);*	Returns the *child* of the given object.
void id.set(newFirstChild);	Sets the object as the first *child* under the same *parent.*
*TYPE *parent,*obj;* *id_iterator it(parent); while(obj= ++it){ ... };*	Traverses all children of the given *parent*. Value of *obj* should not be modified from within the loop. Object can be deleted from the tree while traversing.

Commands available only for the *DOUBLE_TREE*:

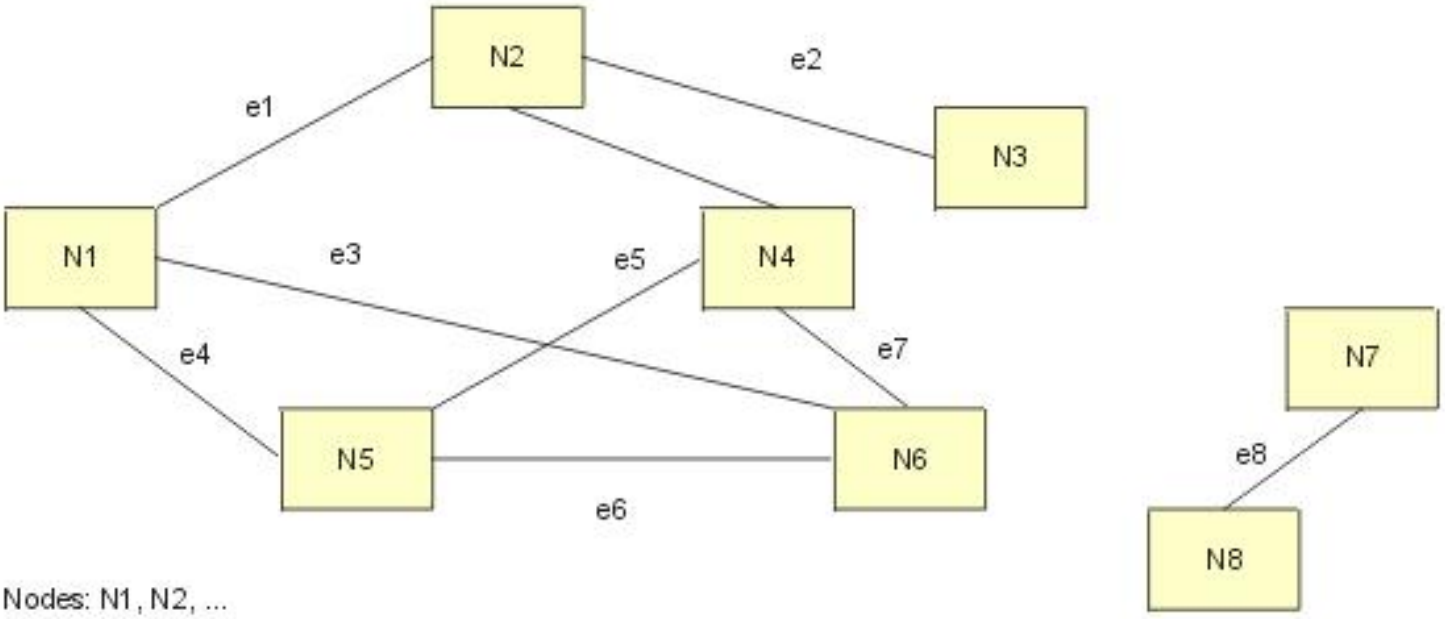
*void id.ins(TYPE *obj,TYPE *newSibling);*	Inserts a new sibling prior to the given object (under the same *parent*).
TYPE id.bwd(TYPE *obj);*	Returns the previous (*bwd=BACKWARD*) *child* under the same *parent.*
*TYPE *parent,*obj;* *id_iterator it(parent); while(obj=it--){ ... };*	Works just like the other loop above, except that it traverses the set in reverse order.

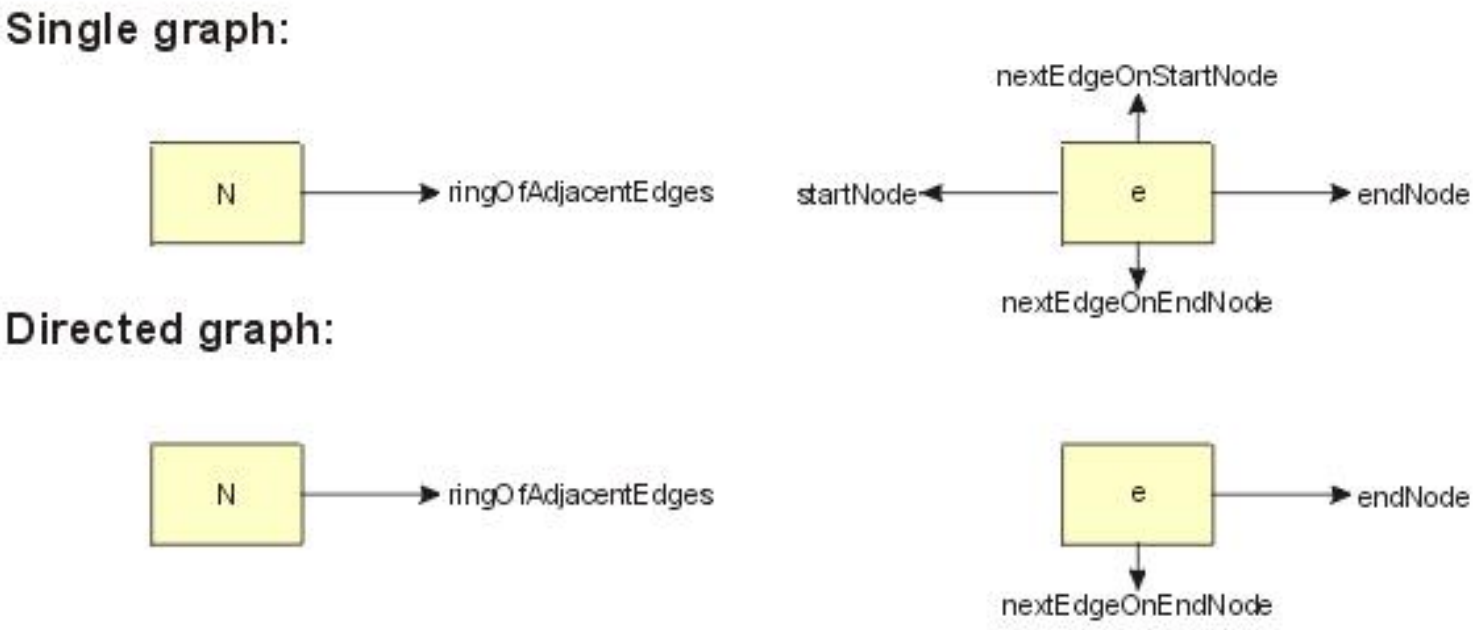
Examples:

1. Calculation of current and wire width in complicated tree-like power networks on a VLSI chip (*test10a.c*).
2. NIAL-like language that manipulates objects and arrays of objects hierarchically arranged in a tree (*test12a.c*).
3. Resetting the order of the subtrees with *set()* (*test10b.c*).

« Previous Section 11.3 AGGREGATION Next Section 11.5 GRAPH »

11.5 GRAPH





Purpose:

Two graph representations are available: *DIRECT_GRAPH* for directed graphs, and *SINGLE_GRAPH* for undirected graphs. A graph consists of a set of nodes connected by edges. All nodes are of one type, all edges of another type.

Both graphs preserve the direction of edges. However, the difference between the two organizations is that, for the *SINGLE_GRAPH*, you can traverse the graph in either direction, while for the *DIRECT_GRAPH* you can proceed only in the direction of the edges.

For example, for a given edge, *SINGLE_GRAPH* gives you access to both adjacent nodes, while *DIRECT_GRAPH* gives you only the target node. Or when traversing edges adjacent to a given node, for *SINGLE_GRAPH*, the *iterator* returns edges both starting and ending at the given node, while the *DIRECT_GRAPH iterator* returns only the nodes starting at the given node.

DIRECT_GRAPH is less expensive in memory; nodes have the same size, but the edge uses only half as many pointers as for *SINGLE_GRAPH*.

In both models the edge rings are singly linked, and therefore the *DELETE* operation is slow in topologies with many edges per node. If you have to deal with situations where edges are deleted frequently, add a doubly-linked version of the graph to the library.

Order of objects:

Edges on a node are currently treated as an unordered collection. However, the order of nodes for each edge is maintained, and determines the direction of the edge. This applies for both directed and undirected graphs. The first of the two nodes is always the *from* node, the second is the *to* node.

Adding edges and nodes:

When you want to add a new edge to the graph, you have to provide two nodes and the edge. If your application uses the direction of the edge, then *np*[0] must be the *from* node, and *np*[1] must be the *to* node:

```
Node *np[2]; Edge *ep;
...
np[0]= ... ; // 'from' node|
np[1]= ... ; // 'to' node
ep=new Edge;
myGraph.add(np,ep);
```

Deleting edges and nodes:

del() deletes a given edge from the graph. If you want to disconnect a node, you have to traverse all its edges, and delete them. This is not easily done for a *DIRECT_GRAPH*. If you want to perform such operations, use a *SINGLE_GRAPH*, and monitor the direction of the edges.

Getting from an edge to adjacent nodes:

For a given *edge* function *nodes(twoNodes,edge)* returns two adjacent nodes. When calling this function,

you give it an array of two pointers; it sets the pointers to the two adjacent nodes. The direction of the edge is preserved (always from *np*[0] to *np*[1]).

```
ZZ_HYPER_SINGLE_GRAPH(myGraph,Node,Edge);
Node *np[2]; Edge *ep;
... /* edge is given */
myGraph.nodes(np,ep); /* returns np */
```

For a directed graph, the use of *nodes()* is exactly the same, except that the function fills in only the target node (*np*[1]). In a directed graph, you cannot reach the source node from the edge.

Moving around the GRAPH:

The *iterator* traverses the edges adjacent to a given node and, through the function *adj()*, provides the adjacent node for each edge. The adjacent node could be found through the function *nodes()*, but when operating in a loop, *adj()* is much faster.

```
id_iterator it(node);
while(edge= ++it){
    adjNode=it.adj();
    ...
};
```

Edges can be deleted while being traversed. For a directed graph, this command traverses all the edges starting at a given node.

For non-directed graph (*SINGLE_GRAPH*), the *iterator* traverses all adjacent edges. If you want to move through the graph in one particular direction, use function *nodes()* which tells you, for each edge, what is the direction of the edge.

There is no arrangement for easy access to all the edges or all the nodes of the graph. Use a separate *RING* if you have to visit all the nodes or edges.

Syntax:

ZZ_HYPER_SINGLE_GRAPH(id,Node,Edge);	Declares the organization, Node and Edge are type names.
*void add(Node *np[2],Edge *e);*	Adds edge *e* between the two given nodes.
*void insert(Node *np[2],Edge *e);*	Just like *add()*, but does not reverse the order of the edges.
*void set(Node *np,Edge *e);*	Sets *e* as the new tail of the edge list.

*void del(Node *np[2],NULL);*	Deletes the edge between the two given nodes.
*void del(Node *np[2],Edge *e);*	Deletes the edge regardless of what nodes are given.
void nodes(Node *np[2], Edge *e);	For the given edge, it fills in pointers to the adjacent nodes.

```
Node *n0, *n1; Edge *e;
id_iterator it(n0); // start from n0
while(e= ++it){ // traverse adjacent edges
    n1=it.adj(); // adjacent node
    ...
 }
```

ZZ_HYPER_DIRECT_GRAPH(id,Node,Edge);	Declares the organization, Node and Edge are type names.
*void add(Node *np[2],Edge *e);*	Adds edge *e* between the two given nodes.
*void insert(Node *np[2],Edge *e);*	Just like *add()*, but does not reverse the order of the edges.
*void del(Node *np[2],NULL);*	Deletes the edge between the two given nodes.
*void del(Node *np[2],Edge *e);*	Deletes the given edge regardless of *np*[1], but the starting point *np*[0] must always be given.
void nodes(Node *np[2], Edge *e);	For the given edge, it fills in pointers to the adjacent nodes.

```
Node *n0, *n1; Edge *e;
id_iterator it(n0); // start from n0
while(e= ++it){ // traverse adjacent edges
    n1=it.adj(); // adjacent node
    ...
 }
```

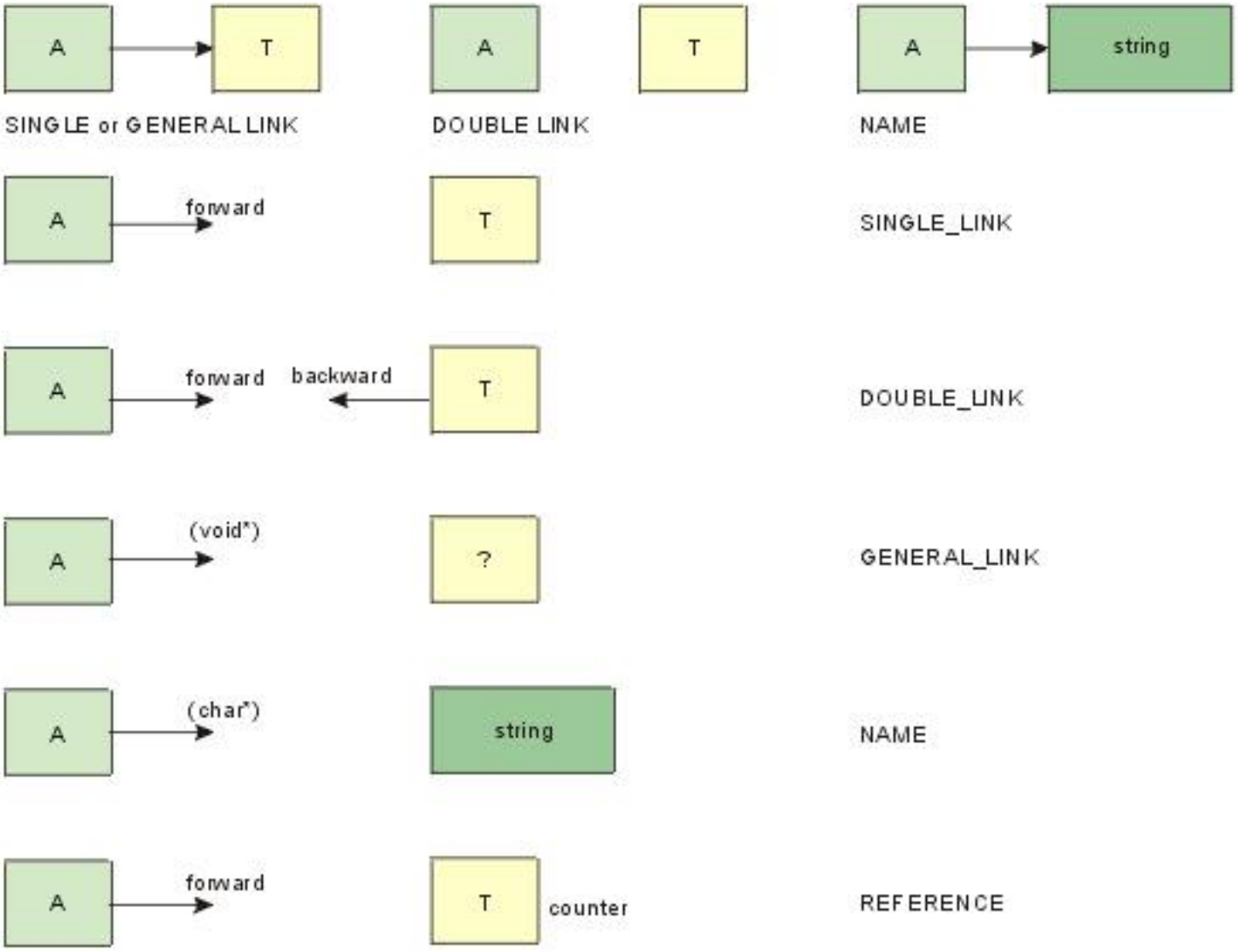
Examples:

1. Two superimposed graphs of highways and airline routes connecting a set of towns (*test14a.c*).
2. Simple test which creates a small tree, using *DIRECT_GRAPH*, and then it deletes a subtree is in *test49a.c.*
3. The same problem as in *test49a.c* but coded with *SINGLE_GRAPH* is shown in *test49b.c*. Note the changes required to the code, because the interators traverse both edges that come to a vertex or leave it.

Previous Section 11.4 TREE

Next Section 11.6 LINK and NAME

11.6 LINK, NAME, and REFERENCE



Purpose:

LINK provides a simple relational link between two objects. *NAME* provides a simple link between an object and a character string, typically a name. An object can have any number of links and names. Each link is equivalent to one pointer. *REFERENCE* provides a simple pointer plus a reference counter on the target object. This can be used for automatic garbage collection, Smalltalk style.

The advantage of using these organizations instead of simple pointers is that when saving/retrieving whole organizations from disk even these simple pointer relations are automatically converted to new values. Any name declared through the *NAME* organization is treated as a separate object. The global *util.save()* automatically saves all names with other objects. However, when saving individual objects with the macro *ZZ_OBJECT_SAVE()*, each name has to be saved individually (see Chap.13).

Five types of LINK:

SINGLE_LINK provides a simple one-way link from structure type A to object type T.

A *DOUBLE_LINK* provides a two-way link between objects of type A and T. Object A is always considered primary and T secondary; *forward* means from A to T, *backward* means from T to A.

A *GENERAL_LINK* is a single link to an object of unknown type. Internally, the pointer is cast as (*char* *). Automatic *SAVE/OPEN* resets this pointer; however, the *SWEEP* operation does not pass through this pointer when collecting objects.

A *NAME* links the object to a character string.

A *REFERENCE* provides a fully typed pointer link from type A to an object of type T. The T object keeps an integer counter, which counts number of reference pointers pointing to it.

Adding a link:

add() adds a new link between two objects. The object types must agree with the definition of the link. For a *REFERENCE*, this operation adds 1 to the count.

Deleting a link:

del() deletes the link. For *NAME* types, just like in other links, no de-allocation is done. The string must be freed by calling *sfree()*. For a *REFERENCE*, this operation substracts 1 from the count.

Jumping across the link:

fwd() moves you from A to T; for *NAME*, it returns the name of the object.

bwd() moves you from T to A (available only for *DOUBLE_LINK*).

Accessing the counter:

When working with a *REFERENCE*, the following commands permit to access the counter:

id.get(T)* returns the current content of the counter,

id.set(T,int)* sets the counter to a given value. Normally, the user should not set the counter. The counter is initialized to 0 for a new object, and the counter is updated automatically by all *add()* and *del()* calls.

SAVE/OPEN:

It may appear strange to use special organizations for such simple pointer relations. The reason is that

when you are saving data to disk (structure blasting), only the pointers registered under OrgC++ are properly restored. Unregistered pointers become invalid (meaningless) when restoring data from disk.

The method of allocating the name string may be important, if you plan to save data to disk later. There are two ways of adding a name to an object:

```
ZZ_HYPER_NAME(name,Employee);
Employee *e;
name.add(e,"Brown J");

ZZ_HYPER_NAME(name,Employee);
Employee *e; char *n;
n=util.strAlloc("Brown J");
name.add(e,n);
```

In the first case, we simply use a string provided by the compiler. In the second case, we allocate a special copy with *malloc*.

When saving to disk, the saving algorithm has to temporarily change some bytes on all objects, including the name strings. However, SUN and some other compilers consider compiler-supplied strings sacred, and if any function attempts to change them, the program core dumps. For this reason, the second approach (using *util.strAlloc()*) should be used if you will be saving data to disk.

The text string stored by the *NAME* organization may contain any characters except for `\0' (end of string character), including the blank (white space) character.

Syntax:

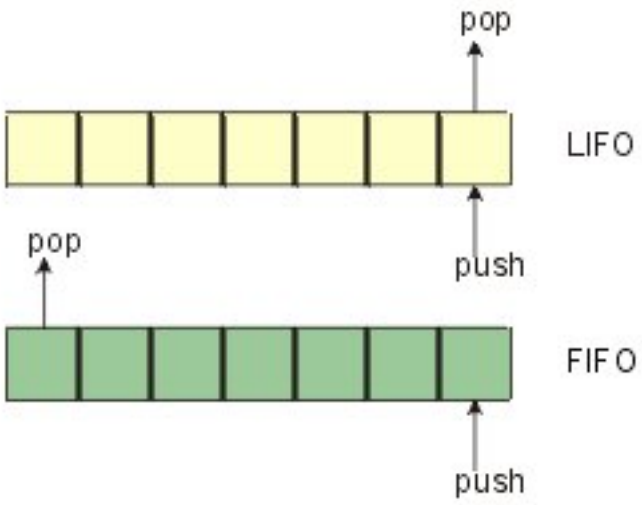
ZZ_HYPER_SINGLE_LINK(id,FROM,TO); *ZZ_HYPER_DOUBLE_LINK(id,FROM,TO);* *ZZ_HYPER_GENERAL_LINK(id,FROM);* *ZZ_HYPER_NAME(id,FROM);*	Declare the three types of links and the *NAME* organization. *FROM* and *TO* are the object types. For the *NAME*, the *TO*-type is *char** by default, for the *GENERAL_LINK* (*void **) is the default.
*void id.add(FROM *a, TO *b);*	Adds a link from *a* to *b*.
TO id.del(FROM *a);*	Deletes the link starting at *a*, and returns a pointer to the deleted object.
TO id.fwd(FROM *a);*	For a given object *a*, it returs the object connected by the link (*fwd=FORWARD*).
FROM id.bwd(TO *b); // for DOUBLE_LINK only*	Returns the source of the link that points to this object (*bwd=BACKWARD*).

Examples:

1. While working with the Rectangle record, *test8.c* assumes that, temporarily, more data such as perimeter and area are required. *LINK* is used to link the Rectangle record with the temporary record, tempRect.
2. When working with two record types, Couple and Person, it may be useful to have a link from a Couple to the two people that form it. In *test16a.c* this is implemented as two *LINK* organizations between Couple and Person.
3. The program shown in *test15a.c* handles Employee records that contain, besides other information, the employee name. The name is linked by a regular pointer, without using the organization of *NAME*. However, when we want to save the same data on disk *(test15b.c*), *NAME* is used instead of the pointer.
4. In *test14a.c*, both States and Towns have names declared through the *NAME* organization.
5. *test19a* and *test19b* illustrate the use of *SAVE* on an organization that involves a *GENERAL_LINK*. For more information on *REFERENCE* see files *orgC/macro/hyprefer* and *test42b.c*.

Previous Section 11.5 GRAPH Next Section 11.7 STACK

11.7 STACK



Purpose:

The *STACK* organization provides fast *LIFO* (last-in first-out) and *FIFO* (first-in first-out) stacks.

There are three ways of implementing a stack in OrgC++:

1. As described previously, *RINGS* can be used as *LIFO/FIFO* queues.
2. Special organizations called *LIFO* and *FIFO* are internally based on *RINGS*, but have convenient *push()* and *pop()* functions.
3. The *DYNAMIC_ARRAY* can be used as a *LIFO* stack. *push()* and *pop()* operations add/delete the top object from the array, and enlarge the array automatically if the array is full.

POP and PUSH operations:

push() pushes a new object onto the stack;
pop() returns the next object and disconnects it from the stack.
When *pop()* returns NULL, the stack is empty.

Initialization:

When using the *RING*, *LIFO*, or *FIFO* organizations, *start*=NULL must be set before using the stack.
When using the array-style stack, the array must be declared by *ZZ_HYPER_ARRAY()*, and formed by *form()* (See Chap.11.15)

Syntax:

ZZ_HYPER_LIFO(id,TYPE); *ZZ_HYPER_FIFO(id,TYPE);*	Declare the two types of the queue.
*void id.push(TYPE *obj);*	Pushes the given object into the queue.
TYPE id.pop();*	Pops another object from the queue. When returning NULL, the queue is empty.

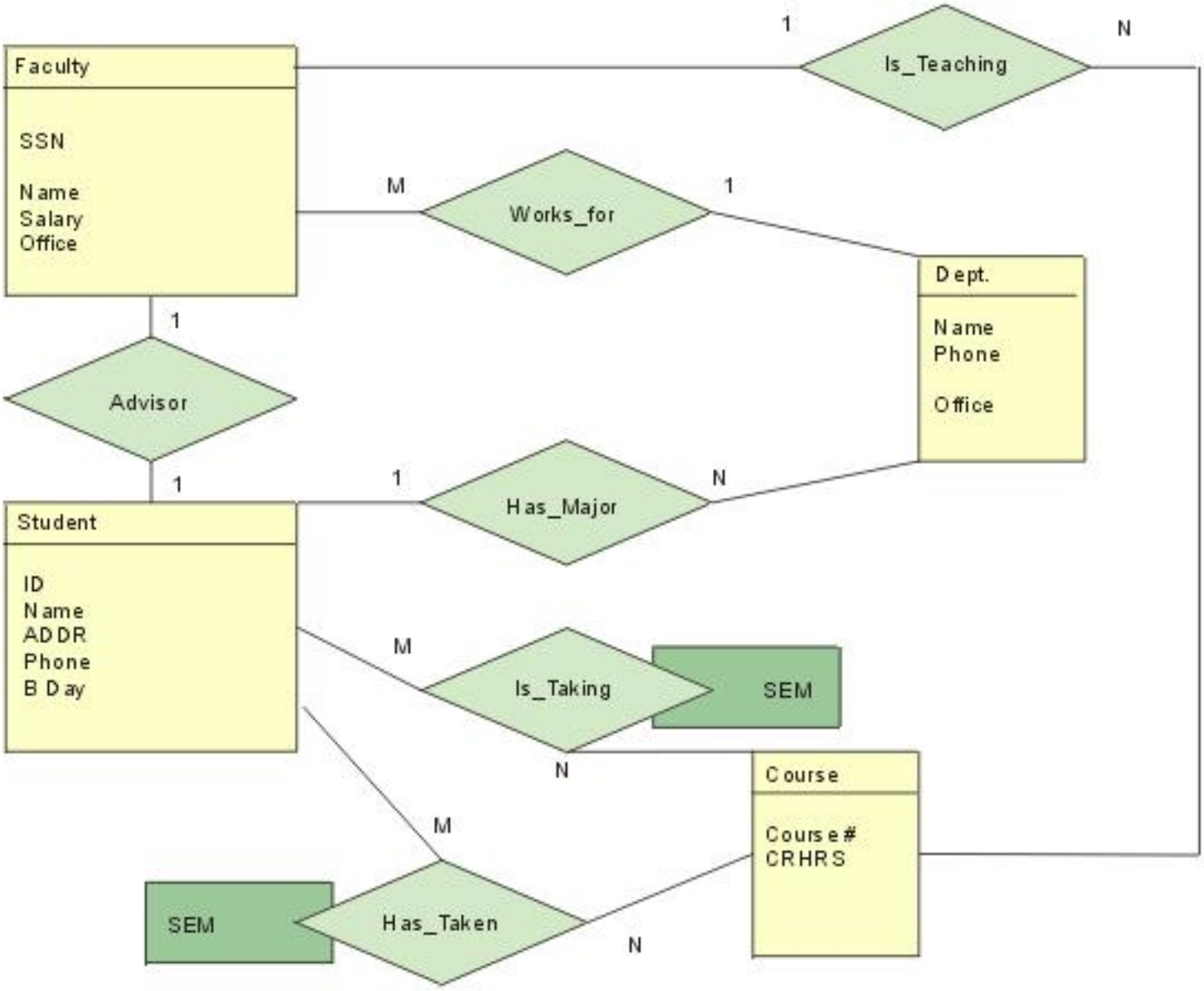
Examples:

1. See the files *macro/hyplifo* and *macro/hypfifo.*
2. *test9.c* contains a comprehensive test of all *ARRAY* features.

Previous Section 11.6 LINK and NAME

Next Section 11.8 ENTITY_RELATIONSHIP MODEL

11.8 ENTITY-RELATIONSHIP MODEL



This figure is an example from a paper by Campbell, Czejdo, and Embley. Rectangles represent entities, diamonds, relations between them, and ovals attributes. Both entities and relations can have attributes.

Purpose:

The four basic organizations (see below) cover all situations that can occur when building a database within the ER model. You declare entities and relations as C++ classes; ER attributes simply become attributes of these structures. All organizations are internally doubly linked, so that the "delete" operation and traversing through the network is fast in all directions.

The Four ER models:

ZZ_HYPER_1_TO_1 (id, sourceEntity, relation, targetEntity);

ZZ_HYPER_1_TO_N (id, sourceEntity, relation, targetEntity);

ZZ_HYPER_M_TO_1 (id, sourceEntity, relation, targetEntity);

ZZ_HYPER_M_TO_N (id, sourceEntity, relation, targetEntity);

The Four ER models:

ZZ_HYPER_1_TO_1 (id, sourceEntity, relation, targetEntity); declares 1-to-1 relation which is similar to *DOUBLE_LINK*, except that it has the *relation* object sitting in the middle of the link.

Using *DOUBLE_LINK* for this purpose is less fancy, but more efficient. The same thing applies to 1-to-N and M-to-1 relations below. Only the M-to-N relation is a special organization in its own right, which has no other simple representation.

ZZ_HYPER_1_TO_N (id, sourceEntity, relation, targetEntity); connects several target entities with one source entity, and is similar to *SINGLE_AGGREGATE* (formerly *SINGLE_TRIANGLE*), except for the relation sitting in the middle. If you don't keep any data on the relation itself, use *SINGLE_AGGREGATE* instead.

ZZ_HYPER_M_TO_1 (id, sourceEntity, relation, targetEntity); connects several source entities with one target entity, and is similar to *SINGLE_AGGREGATE* (formerly *SINGLE_TRIANGLE*), except for the relation sitting in the middle. If you don't keep any data on the relation itself, use *SINGLE_AGGREGATE* instead.

ZZ_HYPER_M_TO_N (id, sourceEntity, relation, targetEntity); connects several source entities with several target entities; the *relation* objects form the connections.

The access to all organizations is generic, as shown in the **Syntax** section.

Error protection:

As a result of strong typing, many coding errors get caught by the compiler. At run time, the organization is automatically protected against dangling pointers. An object must be completely disconnected before it can be freed. You cannot replace a relation using *add();* first the old relation must be disconnected by *del().*

Required memory:

1_TO_1 1 pointer on each entity, 2 on the relation;

1_TO_N 1 pointer on each entity, 4 on each relation;
M_TO_1 1 pointer on each entity, 4 on each relation;
M_TO_N 1 pointer on each entity, 6 on each relation;

Fast name pickup:

In databases based on the ER model, fast pickup of an object's name is usually required. Use the HASH organization in conjunction with the ER model - see the example in *orgC/test/test18a.c*.

Alternate Implementation:

The four basic models permit uniform handling of all ER situations. However, relations that have no attributes can also be implemented as a *DOUBLE_LINK* (1_TO_1 relation) or as a *DOUBLE_TRIANGLE* (1_TO_N and M_TO_1 relations). Such representations use fewer objects and 2 less pointers per relation. The penalty is that some of the uniformity is lost.

Syntax:

ZZ_HYPER_1_TO_1(id,SOURCE,REL,TARGET); *ZZ_HYPER_1_TO_N(id,SOURCE,REL,TARGET);* *ZZ_HYPER_M_TO_1(id,SOURCE,REL,TARGET);* *ZZ_HYPER_M_TO_N(id,SOURCE,REL,TARGET);*	These statements declare the 4 basic relations. *SOURCE* stands for the type of the source entity, *REL* stands for the relation, and *TARGET* for the type or the target entity.
*void id.add(SOURCE *s,REL *r,TARGET*t);*	Adds relation *r* between entities *s* and *t*.
*void id.del(REL *r);*	Disconnects the given relation.
REL id.fwd(SOURCE *s);*	For a given source node, this returns the first relation starting from it (*fwd=FORWARD*).
REL id.bwd(TARGET *t);*	For a given target entity, this returns the first relation leading to it (*bwd=BACKWARD*).
SOURCE id.source(REL *r);*	For a given relation, it returns the source entity.
TARGET id.target(REL *r);*	For a given relation, it returns the target entity.
*SOURCE *s; REL *r;* *id_sIterator is(s);* *while(r=is++){ ... }*	For a given *s*, this loop traverses all relations starting at *s*.
*TARGET *t; REL *r;* *id_tIterator it(s);* *while(r=it++){ ... }*	For a given *t*, this loop traverses all relations that lead to *t*.
is.start(s); *it.start(t);*	Restart the iterators.

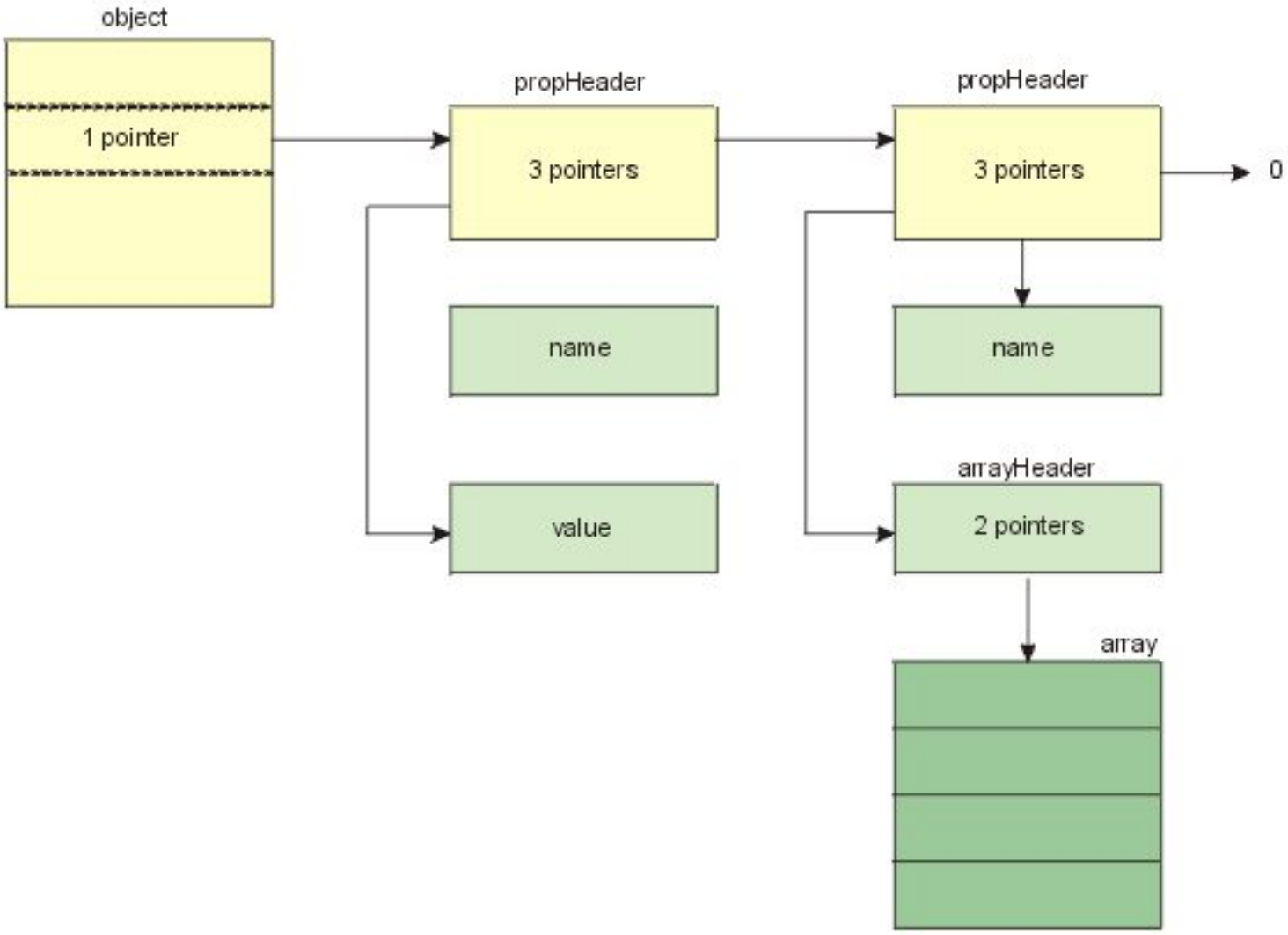
Examples:

1. See files *macro/hyp1to1*, *macro/hyp1ton*, *macro/hypmto1*, and *macro/hypmton*.
2. *test18a.c* contains a comprehensive test of all ER features on the database shown in the figure above.

Previous Section 11.7 STACK

Next Section 11.9 Property (Run-Time Extensibility)

11.9 PROPERTY (RUN-TIME EXTENSIBILITY)



Purpose:

Objects that have PROPERTY may be extended at run time by any number of labelled attributes, called PROPERTIES. A property can be an *integer*, a *float*, a *character*, a *string*, or an array of these.

Property exists on the object itself, it is not associated with a hyper-object. There can be only one set of properties per object.

Memory requirements:

Declaring PROPERTY on an object adds 1 pointer to all objects of this type. If the property is not used, the pointer is NULL. When allocating or initializing objects, this pointer is automatically set to NULL.

Each single value property has an overhead of 2 additional pointers plus the string for the property name. Each array property has an overhead of 5 additional pointers plus the string for the property name.

Saving an object with properties:

save() and *open()* consider properties to be part of the object. Properties are automatically saved/restored from disk with the object.

Available functions/macros:

setProp()	adds a new property. If a property with this name already exists on the object, the value/array is reset. If the type of the property or the time of the array is changed, a warning message is issued.
getProp()	searches properties within a given object, looking for the given property name, and returns its value/array.
prtProp()	is a general utility which, for a given property value, label, etc, prints the property in human readable format. This function is different from the other *PROPERTY* functions in that it belongs in the hyper_class *UTILITIES*.
<type>_propIterator	declares and initializes an iterator for an object of the class type.
start()	restarts a loop on an iterator, which has already been declared.
next()	is an iterator function which moves to the next property on the same object.
delProp()	deletes a property.

Syntax:

Note that all methods are associated with the *PROPERTY*, except for *prtProp()*, which is a part of *UTILITIES*, because it does not depend on property, objects, or anything else. It is just a generic printing function.

// type is the property type: int, float, char,string *class TYPE;* *ZZ_HYPER_PROPERTY(TYPE);* *ZZ_HYPER_UTILITIES(util);* *void TYPE::setProp(char *type,char *propName,char*val[],int size)*;	adds/resets property with name *propName* on *obj*. The value of the property must be in *val*, whether a single value or an array. *size=1* for a single value, >1 for an array.
*char** TYPE::getProp(char *propName,char **type,int*size)*;	for a given object and property name, it returns the property value/values (as in array *val*) and then, through parameters, returns the property *type*, and the size of the array. *size=0* means the property was not found.

*void TYPE::delProp(char *propName);*	deletes the property with the given name.
*void util.prtProp(FILE *fp,char *type,char*propName,char *val[],int size);*	prints the given property as received for example from *getProp()*. It prints this into file *fp* in human readable format.

TYPE_propIterator it(obj); // initiates iterator
it.start(obj); // restarts the iterator on obj
*char** it.next(char *propName,char **type,int *size)*
	returns the array value (*val*), and associated attributes, and can be used to loop through all properties on one object. It returns NULL after the last property. For example:
	`// printing all properties on obj`
	`type_propIterator it(obj);`
	`while(val=it.next(&propName,&type,&size)){`
	`util.prtProp(fp,propName,type,val,size);`

Examples:

Note that the syntax of functions working with *PROPERTY* is quite different in C and C++.

1. Look at the test which adds various types of properties to an object of type *Apple* (*test4d.c*). This program also issues a warning message when the property type is changed. If you are used to the C version of Organized C, compare this program with its C equivalent in *test4b.c*.
2. Another C example is the NIAL-like language in *test12a.c* which works with objects organized in a tree. Each object can have an atom - which is a single value/array-like attribute. In *test12a.c* atoms are implemented as *PROPERTY*. Sorry, this example is not yet available in C++.

« Previous Section 11.8 Entity-Relationship Model

Next Section 11.10 Run-Time Type Detection »

11.10 SELF_ID

Purpose:

In some projects, you may need a pointer that can point to several different object types (*GENERAL_LINK*). When you arrive at an object via such a pointer, you usually want to know the type of that object. In C, our library provided the *SELF_ID* organization for this purpose.

Using a typeless pointer is not a good strategy in C++ and should generally be avoided except for special situations. If you want to detect the object type, the standard method is to derive all of the classes (or at least those that you want to detect) from the same base class, and use virtual functions. For example:

```
class Obj {
    ...
    virtual char *Iam(void){return "Obj";}
};
class Automobil : public Obj {
    ...
    virtual char *Iam(void){return "Automobil";}
};
class Airplane : Obj {
    ...
    virtual char *Iam(void){return "Airplane";}
};
```

In tests *test4d.c* and *test7a.c*, the original *SELF_ID* organization has been replaced with this more elegant technique.

However, this method has a serious disadvantage. If you attempt to call function *Iam()* on an object which is not derived from class *Obj*, your program will crash. The technique is not useful for sorting out incorrect (garbage) objects from correct types.

Chap.11.15 describes how OrgC++ gives you access to internal type tables, normally hidden by the compiler. This information can be also used for a safer type detection. In order to make this simple for you, the *TYPE* organization provides function *vfCheck(void *p)* which, for a pointer to any memory location, can tell you whether the object is of the type registered with *ZZ_EXT_*.., and what the type is.

```
ZZ_HYPER_TYPE(type);
int tp; // type index
void *p;
...
tp=type.vfCheck(p);
```

Function *vfCheck()* provides a smart check, because it does not require the existence of the common root object, and it does not crash when inquiring about an unregistered object. I just returns -1 as the signal that the type does not match any registered type.

Function *vfCheck()* does not use virtual function, it compares internal vf pointers used by the compiler, and derives the type from their values.

In order to detect the type, the following conditions must be met:

1. Each detectable type must have at least one virtual function (perhaps just a dummy virtual function).
2. No virtual classes must be used for detectable types.
3. The virtual function pointer must be located at the beginning of the object (this is true for most existing compilers such as Borland 4.0+, Microsoft 8.0+, or SUN 3.0+, but not for SUN 2.1 or Watcom 10.0).

When condition (1) is not met, *vfCheck()* returns -1 (no known type). When conditions (2) or (3) are not met, *vfCheck()* returns -2 (compiler not suitable for type detection). *vfCheck()* never crashes, even when given a pointer to a location which does not represent a valid object.

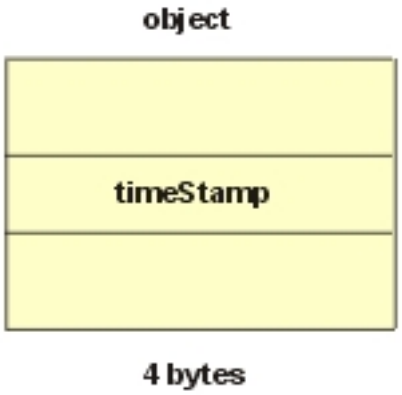
The probability of not detecting a correct type, or of detecting a wrong type is 0. The probability of not detecting an invalid type is very small n/(2**32), where n is the number of valid types.

For an example of *vfCheck()*, see *test39a*.

Previous Section 11.9 Run Time Extensibility

Next Section 11.11 Time Stamp

11.11 Time stamp



Purpose:

To provide a time stamp, which records (with the accuracy of 1 second) the time and date, when the object was created or modified.

In C++, the time stamp is not automatically initialized when creating a new object, unless you place macro *ZZ_INIT()* or call function *setTime()* in all constructors. For example:

```
class myObj { // first variantD
    int i,k;
    ZZ_EXT_myObj
public:
    myObj(){this->setID();}
    myObj(int ii, int kk){this->setID("myObj",1); i=ii; k=kk;}
};
ZZ_HYPER_TIME_STAMP(myObj);
class myObj { // second variant, covers two initializations
    int i,k;
    ZZ_EXT_myObj
public:
    myObj(){ZZ_INIT(myObj);}
    myObj(int ii, int kk){ZZ_INIT(myObj); i=ii; k=kk;}
};
ZZ_HYPER_TIME_STAMP(myObj);
ZZ_HYPER_SELF_ID(myObj);
```

TIME_STAMP is one of the special hard-wired organizations that can have only one instance on any given object. Though it is declared as

ZZ_HYPER_TIME_STAMP(objType);

it does not create a hyper object. Functions manipulating this organization are associated with the object itself, and not with a general id.

Internally, the stamp is packed into 4 bytes. For a convenient comparison of two time stamps, the special function *cmpTime()* is available.

getTime() returns the time stamp unpacked in 6 bytes, each representing one number (year, month, day, hour, minute, second).

Note that the time stamp is kept only on specified object types (those for which *ZZ_HYPER_TIME_STAMP()* have been declared).

Syntax:

// TYPE is the type of the given object *ZZ_HYPER_TIME_STAMP(TYPE);* *char ts[6];* *void TYPE::setTime(void);*	sets the time stamp on the given object to the current time.
void TYPE::getTime(ts);	returns time stamp of the given object [YMDHMS] in ts. For example, June 18, 1989 17:30:11 is represented as [89 6 18 17 30 11].
*int TYPE::cmpTime(TYPE *obj2);*	compares two objects and returns int i<0 when obj1 is older, i=0 when both objects have the same age, i>0 when obj2 is older. Example of use: i=obj1->cmpTime(obj2).

You don't have to worry about the year 2000 and up. The time stamp can be set only to the current date.

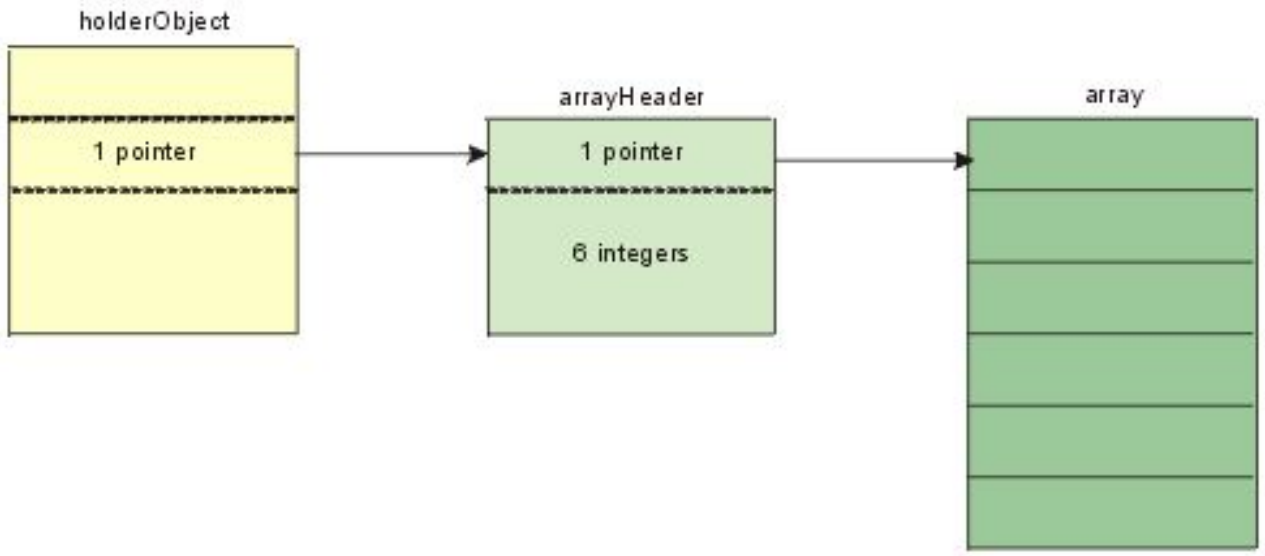
Examples:

1. *test4d.c*

« Previous Section 11.10 Run-Time Type Detection

Next Section 11.12 Dynamic Array and Binary Heap »

11.12 DYNAMIC ARRAY, STACK, BINARY HEAP, SORTED COLLECTION



Purpose:

The prime purpose of this class is to provide an array that automatically reallocates itself when a larger size is required. A BINARY HEAP is only a special case of the ARRAY organization. Note that an ARRAY (or a HEAP) can contain either whole objects or pointers to objects.

In OrgC, an array always exists as an attribute of a holder object. This allows you, for example, to have a ring of objects with an array on each object, or an array of arrays. If you need just an array as such, declare a dummy "holder" object for it.

Memory considerations:

Internally, the holder object contains a pointer to a transparent header, which then points to the array itself. This represents an overhead of 32 bytes under UNIX or 16 bytes for medium/small model under DOS, for each array used. When an array is declared but not used, only one pointer is used.

ZZ_HYPER_ARRAY()	declares the array and the types of objects involved; it neither specifies the size of the array nor allocates space for it.
form()	forms the initial array, and sets the current high-water-mark to -1.
formed()	is useful for checking whether the array has been formed; it returns NULL if it has not been.
reset()	resets the high-water-mark and the increment, but leaves the size intact; no reallocation is done.

size()	returns the current size and high-water-mark. High-water-mark is the highest index actually used so far.
free()	deallocates the whole array. Before freeing an array, all its elements must be disconnected from other data organizations, just like freeing an individual object.

Note that increment>0 represents an arithmetic increment, increment<0 represents a multiplicative increment. For example, increment=-3 will triple the size of the array any time the array size overflows. Increment= -1 is interpreted as an array of fixed size. Index overflow for such an array generates an error message.

Warning:

When an array reallocates itself, the old array becomes free but may not be very useful. For example, when doubling the array size several times, almost 1/2 of the memory may essentially be wasted. Intuitively, an array looks like a simple and practical data arrangement but, in many practical situations, a ring is better; it requires less memory and is faster to operate upon.

Warning:

Array of (*void* *) pointers is not permitted in the current version. Use (*char* *) pointers instead.

Index access (in-line function):

Other objects must refer to the elements of the array by index, never by pointer. However, when indexing an array, a pointer may be used temporarily. For example, *ind()* checks the index range, and returns a pointer to the array. If the index overflows the size, the array automatically resets its size.

*p=id.ind(hp,i); *p=k;* is equivalent to *a[i]=k;*

Inside algorithms, where high speed is required, and the size of the array does not change, you must first establish the base of the array by calling: *a=id.ind(hp,0).* Then you can access the array as if no special management were in force: *a[i]=k; a[12]=a[1]+a[3]* and so on.

Note that *id.head(hp);* is equivalent to *id.ind(hp.0)*; but faster.

Sorting:

An array can be sorted. The user must provide a function that compares two objects. Internally, the sort is based on *qsort().*

Stack:

An array may be used as a stack. After forming the initial array, each *push()* adds one object to the top of the array, and each *pop()* returns the top object of the stack. If the size of the stack overflows the size of the

array, the array increases its size by the given increment.

Sorted Collection:

In addition to function *sort()* which sorts the array, there is aset of functions which keep the array sorted even after adding or removing its elements. Such sorted array can be used as an ordered collection. This applies not only to arrays of full objects, but also to arrays of pointers.

Operations on sorted arrays are relatively fast: *sort()* uses the system function *qsort()*, binary search is used to find objects in the array, and a fast system function moves blocks of memory when a inserting or removing an element of the array.

If you want to keep the array sorted, restrict yourself to using the following commands: *addOrd(), delOrd(), getOrd(), ins()*, and *ind().*

For an example of using a sorted array, see *test53.c.*

Binary Heap:

A partially sorted binary heap can be efficiently implemented as an array. The heap requires a function which compares two array entries, just like the function required for *qsort().* The heap is always ordered so that the minimum array entry is at the top of the heap (index=0).

inHeap()	inserts a new object into the heap;
outHeap()	returns the top of the heap and removes it from the heap;
delHeap()	deletes an object from the middle of the heap;
updHeap()	re-sorts the heap when an object in the middle of the heap changes its value.
size()	can be used to detect an empty heap.
delHeap() and *updHeap()*	allow you to use the heap as a full priority queue, where objects already sitting in the queue may change their priority or be deleted from the queue.

All HEAP operations require that a callback function is given. Any time an object changes its position within the heap, this function is called with the new index location. This permits you, for objects outside of the queue, to monitor permanently the position of the objects within the queue. This can be best seen from the following example.

Example:

```
class Head {
    ZZ_EXT_Head
};
```

```
class Employee {
    ZZ_EXT_Employee
public:
    int heapIndex; /* everything public just for demonstration */
    int salary;
    ...
};
class PtrEmpl {
    ZZ_EXT_PtrEmpl
};
ZZ_HYPER_ARRAY(heap,Head,PtrEmpl);
ZZ_HYPER_SINGLE_LINK(link,PtrEmpl,Employee);
void bck(void *p,int i){ /* call-back function to record position */
    Employee *e;
    e=link.fwd((PtrEmpl *)p);
    e->heapIndex=i;
}
int sortFun(const void *v1,const void *v2){
    Employee *e1,*e2;
    e1=link.fwd((PrtEmpl *)v1);
    e2=link.fwd((PtrEmpl *)v2);
    return(e2->salary - e1>>salary);
}
Head *h;
PtrEmpl *p;
Employee *e;
int i;
...
/* assume Employee e is given and should enter the heap */
p=new PtrEmpl;
link.add(p,e);
heap.inHeap(sortFun,h,p,bck); /* automatically updates heapIndex */
...
/* now we can update any Employee object */
e->salary=4600; /* new value */
i=e->heapIndex;
if(i>=0)heap.updHeap(sortFun,h,i,bck);
```

The parameters of the callback function are:
*void *p;* ... pointer to the object in the queue, cast as (*void**);
int i; ... index in the queue.

If the object in the queue provides a pointer to the outside object, this is enough to reach the outside object. When using only *inHeap()* and *outHeap()*, this function is not really needed, and can be replace by a

dummy function

*void bck(void *p,int i){}.*

Using a heap of pointers is usually better than working with a heap of whole objects.

Arrays and pointers:

Generally, it is not good practice to access elements of an array via pointers. An array is a sequential arrangement of objects, and should be accessed by index. If you decide to violate this rule (the reason may be the performance of some special algorithm), make sure that the array has a fixed size (form the array with increment= -1). If the array may reallocate itself, pointers leading into the array may suddenly become invalid.

There are three typical situations when pointers leading into an array may cause trouble:

1. Temporary object linked to the array:

```
struct tempObj {
    ....
    Blob *b;
};
ZZ_HYPER_ARRAY(ar,Head,Blob);
Head *hp; tempObj t;
    ...
t.b=ar.ind(hp,42);
```

2. Remembering a pointer for too long

```
typedef struct Blob Blob;
class Blob {
ZZ_EXT_Blob
public:
    int i;
};
ZZ_HYPER_ARRAY(ar,Head,Blob);
Head *hp; Blob *b;
    ...
b=ar.ind(hp,42);
b->i=31; /* perfectly fine */
    ... /* lot of array manipulation */
b->i=32; /* may crash the run */
```

3. Transparent pointers

```
ZZ_HYPER_ARRAY(ar,Head,Blob);
ZZ_HYPER_SINGLE_RING(myRing,Blob);
```

The array must be fixed, otherwise reallocation of the array may destroy the ring.

Sometimes, a change of organization can improve the robustness of the data. In the following organization, the AGGREGATE parent pointer leads from Employee to Department:

```
class Header {...};
class Department {..};
class Employee {... };
ZZ_HYPER_ARRAY(myArray,Header,Department);
ZZ_HYPER_SINGLE_AGGREGATE(inDept,Department, Employee);
...
// potential problem with the parent pointer in 'inDept'
a=myArray.form(hp,2000,500);
...
// no problem but what do you do if the array is not big enough
a=myArray.form(hp,2000,-1);
```

Better organization is to store an integer index directly inside the Employee class, and update it manually:

When no pointers lead into an array, there is no potential danger or restrictions:

```
class Header {...};
class Department {..};
class Employee {int parent; ... };
ZZ_HYPER_ARRAY(myArray,Header,Department);
ZZ_HYPER_SINGLE_COLLECT(inDept,,Department,Employee);
```

Saving to disk:

save() and *open()* treat an ARRAY as a part of the holding object, and they save/restore the array automatically with it. In the *SWEEP* operation, the expansion process passes through the array and its elements as if they were normal single objects.

Whether you use a plain C++ array, or the dynamic ARRAY from the OrgC++ library, make sure that *#define ZZ_INHERIT* is included in your *environ.h* file.

If you have an array of (*char* *) pointers, the pointers will be properly updated after *save()* and *open()*. However, the expansion algorithm will not expand through these pointers. The situation is similar to the use of *GENERAL_LINK* (see p.11.6.2).

Syntax:

class Holder; *class Element;* *ZZ_HYPER_ARRAY(id,Holder,Element);* *typedef int (*sortFun)(const void*,const void*);* *// compares two objects as required for qsort* *typedef int (*callBack)(void*,int);* *// provides feedback about the position in the heap* *Element* id.form(Holder *hp,int size,int incr);*	forms an array on hp, and returns a pointer to it for fast access.
void id.formed(Holder *hp);*	returns NULL if the array has not been formed.
*void id.free(Holder *hp);*	frees the array identified as id on holder hp.
int id.size(hp,&hWater,&incr);	returns the current size and, through parameters, high-water-mark and increment.
Element id.ind(Holder* hp,int index);*	checks index range, and returns pointer to the appropriate element of the array.
*Element * id.head(Holder* hp);*	same as id.ind(hp,0) but faster.
*void id.reset(Holder *hp,int hWater,int incr);*	resets the high_water_mark and increment to new values.
*void id.sort(sortFun ff,Holder *hp);*	sorts the array using *ff* for comparison of objects.
*void id.push(Holder *hp,Element *p);*	pushes Element *p* onto the array which is being used as a stack.
Element id.pop(Holder *hp);*	pops the next Element from the array which is being used as a stack.
*void id.inHeap(sortFun ff,Holder *hp,Element*p,callBack bck);*	pushes Element *p* onto the heap. Function *ff* controls the heap.
Element id.outHeap(sortFun ff,Holder *hp,callBack bck);*	returns the top element, and removes it from the heap
*void id.updHeap(sortFun ff,Holder *hp,int index,callBack bck);*	resorts heap starting from the given index.
*void id.delHeap(sortFun ff,Holder *hp,int index,callBack bck);*	deletes element at the given index.
*void addOrd(sortFun ff, Holder *hp, Element *p);*	adds *p* to a sorted array, maintaining the sorted order.

*void delOrd(sortFun ff, Holder *hp, Element *p);*	finds *p* in the sorted array, and removes it, maintaining the order.
*void delOrd(sortFun ff, Holder *hp, int ind);*	deletes element with index *ind*, while maintaining the order.
*int getOrd(sortFun ff, Holder *hp, Element *p, int*found);*	using binary search, finds the index of the element which has the same key as element *p*. If a match is found, the function returns the index, and sets *found=1*. If the match is not found, the function returns the next of the element, before which *p* would have to be inserted in order to maintain the order, and sets *found=0.*
*void ins(Holder *hp, int ind, Element *p);*	copies element *p* into position *hp*, shifting the right section of the array to the right. This function should be used only after checking with *getOrd()* where to insert the new element; without being careful, it may destroy the order of the array.When adding new elements to the array, use *addOrd()*; it is faster, and safe.

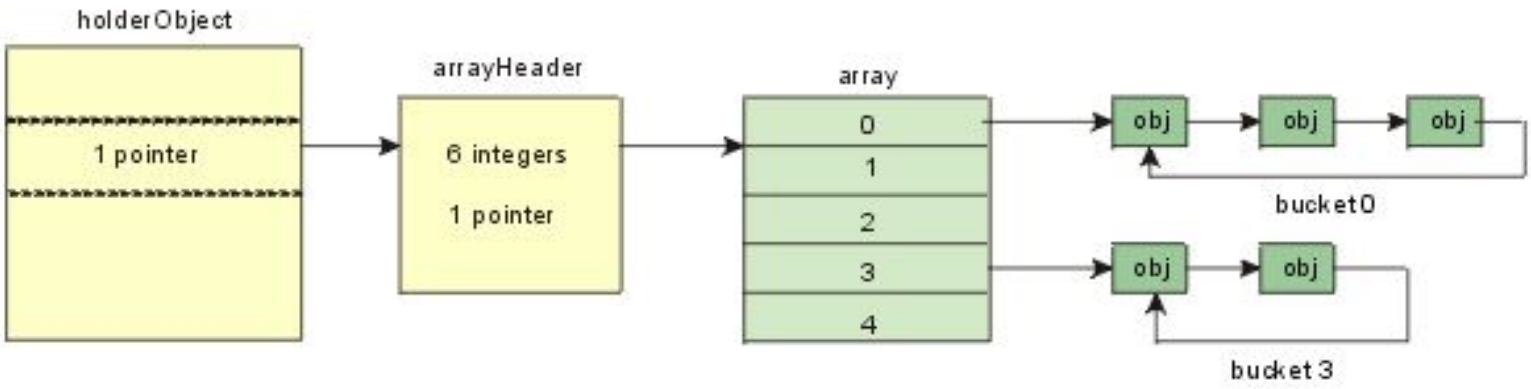
Explanation:

High_water_mark is the highest index used. It is -1 before the array has been used, and equal to *size-1* when it is full. For example, if you formed an array of 10 elements with increment=30, and you called the *ind()* function for index=13,22,58,16, and 0, then the *size()* function will return: size=80, increment=30, and high_water_mark=58.

Examples:

1. A comprehensive test of all ARRAY features is in *test9a.c.* Note that both an array of objects and an array of pointers to these objects are used. Binary heap operations are also tested.
2. *test16c.c* shows how to save an array of objects to disk, and how to work with an array of integers. For the purpose of testing, the data is first saved in ASCII, restored, saved in binary, restored, and only then the third copy is printed.
3. *test32a.c* and *test32b.c* show some examples of advanced handling of arrays.

11.13 HASH TABLES, INDEXED TABLES



Purpose:

Provides fast access to objects either by name or by numerical key. If properly used, a hash table is faster than binary search.

The diagram above shows the internal arrangement, normally transparent to the user. Since the hash table is based on an array, it is attached [just like an array] to a holder object. The array contains pointers to individual buckets, implemented as singly-linked rings. Note that OrgC++ uses no special objects for the bucket. The rings are formed by pointers, which are stored directly inside the objects (under the *ZZ_EXT_*..statement), one pointer per object.

A hash table is treated as a fixed-size array. If you want to change the size, you have to call *resize()*.

The performance of a hash table depends on the algorithm which converts a key into an index. Sometimes, the key may be a combination of several names or numbers. To provide flexibility, OrgC++ requires that, for each hash table, the user provides two functions:

*inline int id_class::cmp(oType *obj1,oType *obj2)* which compares two objects of type *oTYPE* (returns 0 if they have the same key). It also controls the order of the objects inside the buckets (see Indexed tables below.)

*inline int id_class::hash(oType *obj,int size)* which, for a given object and the hash table size, generates the bucket index.

Note that both functions are NOT assigned to class *oType*, but rather to the class that internally represents the hash organization. Normally, except for this case, you don't even know the name of this class, you only deal with its instance, *id*.

If you don't have any special requirements, you can use the default functions provided by OrgC++:

*int ZZhashStr(char *str,int size)* for a given text string and a hash table size, generates the bucket index;

int ZZhashInt(int i,int size) for a given integer and a hash table size, generates the bucket index.

The default functions *ZZhashStr()* and *ZZhashInt()* are based on the multiplicative method (Knuth, Standish), using the improved golden ratio. *ZZhashStr()* first converts the text string into an integer, and then calls *ZZhashInt()* to randomize the number. The conversion of the text into an integer is based on adding individual bytes multiplied by an increasing factor, which generally leads to different indices for strings ABC, CBA, and BCA.

The following example uses two hash tables on the object *Employee*, one based on a character string (name), and the other on an integer key (IDnum). Programs *test25b.c* and *test25c.c* show these hash tables used together with other organizations.

```
#define HASH_SIZE 200
class Header { // header for the hash table
    ZZ_EXT_Header
    ...
public:
    Header();
};
class Employee {
    ZZ_EXT_Employee
    int IDnum;
    char *name;
    ...
public:
    char *getName(void){return name;}
    int getID(void){return IDnum;}
    void setName(char *n){name=n;}
    void setID(int i){IDnum=i;}
};
ZZ_HYPER_HASH(byIDnum,Header,Employee);
ZZ_HYPER_HASH(byName,Header,Employee);
Header::Header(){
    // for each header, form hash tables automatically
    byIDnum.form(h,HASH_SIZE);
    byName.form(h,HASH_SIZE);
    if(util.error()){printf("cannot allocate hash tables\n");
    }
int main(void){
    Employee *p,e;
    Header *h
    h=new Header;
    ...
```

```
    // use hashing without worrying about details
    byIDnum.add(h,p);
    byName.add(h,p);
    // when searching for an object, set up a dummy
    // object with an attribute for which you search
    e.setID(127);
    p=byIDnum.get(h,&e);
    e.setName("Brown J.");
    p=byName.get(h,&e);
    ...
}
// Each hash table requires two functions that control hashing //
    ZZhashStr() and ZZhashInt() are default functions provided
inline int byName_class::cmp(Employee *p1,Employee *p2)
    {return(strcmp(p1->getName(),p2->getName()));}
inline int byName_class::hash(Employee *p,int size)
    { int ZZhashStr(char *name,int size);
    return(ZZhashStr(p->getName(),size)); }T
inline int byIDnum_class::cmp(Employee *p1,Employee *p2)
    {return(p1->getID() - p2->getID()); }
inline int byIDnum_class::hash(Employee *p,int size)
{ int ZZhashInt(int key,int size);
    return(ZZhashInt(p->getID(),size)); }
```

Typical sequence of calls:

- Declare the table with *ZZ_HYPER_HASH()*
- Form the table and give it a size, using *form()*.
- Access the table:
  add() adds an object to the table;
  get() returns an object for a given key;
  add2() adds an object with more control.
- The program can monitor the loading of the table, using *size()*, and when the loading exceeds a certain limit, it can automatically re-size the table.
- *resize()* allocates a new hash table (larger or smaller), re-loads all objects, and frees the old table.
- If you don't need the hash table any more, free it with *free()*.

Indexed tables:

In addition to its basic operation with pseudo)random access of buckets, the HASH class can be also used as an indexed, sorted dictionary. For example, assume that we want buckets sorted by the first letter of each word, and words alphabetically sorted within each bucket. In this case, function *hash()* does not provide random hashing. Instead, it uses the first character as the bucket index.

Function *cmp()* detects identical objects as for the random hashing, but it is also controls the order of objects within individual buckets, using the same syntax as the compare function required for !!TODO Ask Jiri???? or when sorting collections or other OrgC++ organizations. The following code sample demonstrates this approach. For the complete program, look at *test51.c*.

```
class Root {
    ZZ_EXT_Root
    ...
};
class Word {
    ZZ_EXT_Word
    ...
};
ZZ_HYPER_HASH(dict,Root,Word);
ZZ_HYPER_NAME(word,Word);
ZZ_HYPER_UTILITIES(util):
inline dict_class::cmp(Word *w1,Word *w2){
    return strcmp(word.fwd(w1), word.fwd(w2));
    }
inline int dict_class::hash(Word *w,int size){
    char* p=word.fwd(w);
    unsigned char c=(*p); // first character
    int i=(int)(c)(unsigned char)('a'));
    if(i<0 || i>=size)... // error
    return i;
    }
unsigned char ua,uz;
ua='a'; uz='z';
dict.form((int)(uz,ua));
```

Saving to disk:

When saving/retrieving an object to/from disk, a hash table is treated as part of the holder object, and automatically stored with it. The *SWEEP* operation, when collecting all objects, expands through the hash table in the same way it does with pointers and arrays.

When restoring data from the disk, the same hashing function is automatically assumed. If you changed the hash function after saving data to disk, call *resize()* after restoring data from the disk. This will reload the table using the new hashing function.

More than one hash() function

Under special circumstances, you may want to switch between two or more hashing functions while running your program. The two or more optional hashing functions must be included in function

id_class::hash(), using an external switch. For example:

```
class Block {
    ZZ_EXT_Block
public:
    static int hashControl;
    ...
};
class Label {
    ZZ_EXT_Label
public:
    char *name;
    ...
};
int Block::hashControl=0; // controls hashing function
ZZ_HYPER_HASH(bHash,Block,Label);
...
int main(){
... // using hashControl=0 Block::hashControl=1;
bHash.newFun(bp); // resets the hash table 'in place'
}
... // now working with hashControl=1
int bHash_class::hash(Label *p,int size){
    if(!hashControl) return ZZhashStr(p>name,size);
    else return ZZhashStr(&(p>name[1]),size);
}
```

Note how hashControl changes the hashing function from hashing the full name to hashing only a part of the name, starting from the second character. For another, similar example, look at *test25b.c*.

Syntax:

class HOLDER; *class OBJECT;* *ZZ_HYPER_HASH(id,HOLDER,OBJECT);* *void id.form(HOLDER *hp,int size);*	forms a hash table of the given size on the holder object *hp*.
void id.formed(HOLDER *hp);*	allows to check whether the the hash table has been formed; returns NULL if it has not been.
*void id.resize(HOLDER *hp,int size);*	resets the table to a new *size*, reloads all data, and releases the old table.

*void id.add(id,HOLDER *hp,OBJECT *p);*	adds object *p* to the table; if the same key already exists, the object is not added;
*void del(HOLDER *hp,OBJECT *p);*	deletes object *p* from the table;
*int id.add2(id,HOLDER *hp,OBJECT *p,int sameKey);*	adds object *hp* to the table; when sameKey=1 identical keys are permitted, when sameKey=0 they are not; when the object is not added *add2()* returns 1;
OBJECT id.get(HOLDER *hp,OBJECT *p);*	returns the object that has the same key as *p*.
*int id.size(HOLDER *hp,int *totNum);*	returns the current *size* and (through the second parameter) the total number of objects in the hash table.
OBJECT id.slot(HOLDER *hp,int slot);*	returns the first object in the given *slot*.
*id_iterator it(OBJECT *s);*	declares an iterator, and initializes for traversing objects in the same bucket as object *s*.
*void it.start(OBJECT *s);*	initializes previously declared iterator to start another loop.
*void id.free(HOLDER *hp);*	deallocates the whole table, including the array header.
*void id.newFun(HOLDER *hp);*	without reallocating the array, it resets the entire hash table. Typical use: Changing the *hash()* or *cmp()* functions.

Use the following double loop to traverse all objects in the table:

```
int i,size,totNum;
HOLDER *hp;
OBJECT *s,*p;
...
size=id.size(hp,totNum);D
for(i=0; i<<size; i ++){
    s=id.slot(hp,i);
    id_iterator it(s);
    while(p= ++it){
    .. here you have object p ..
    }
}
```

IMPORTANT: Check the error condition after *form()*, or *resize()* by calling *util.error()*. If it is not 0, there is an allocation problem.

Examples:

1. *test25b.c* for fast access of nodes in a graph, searching by name.

« Previous Section 11.12 Dynamic Array and Binary Heap

Next Section 11.14 Pager »

11.14 PAGER

Purpose:

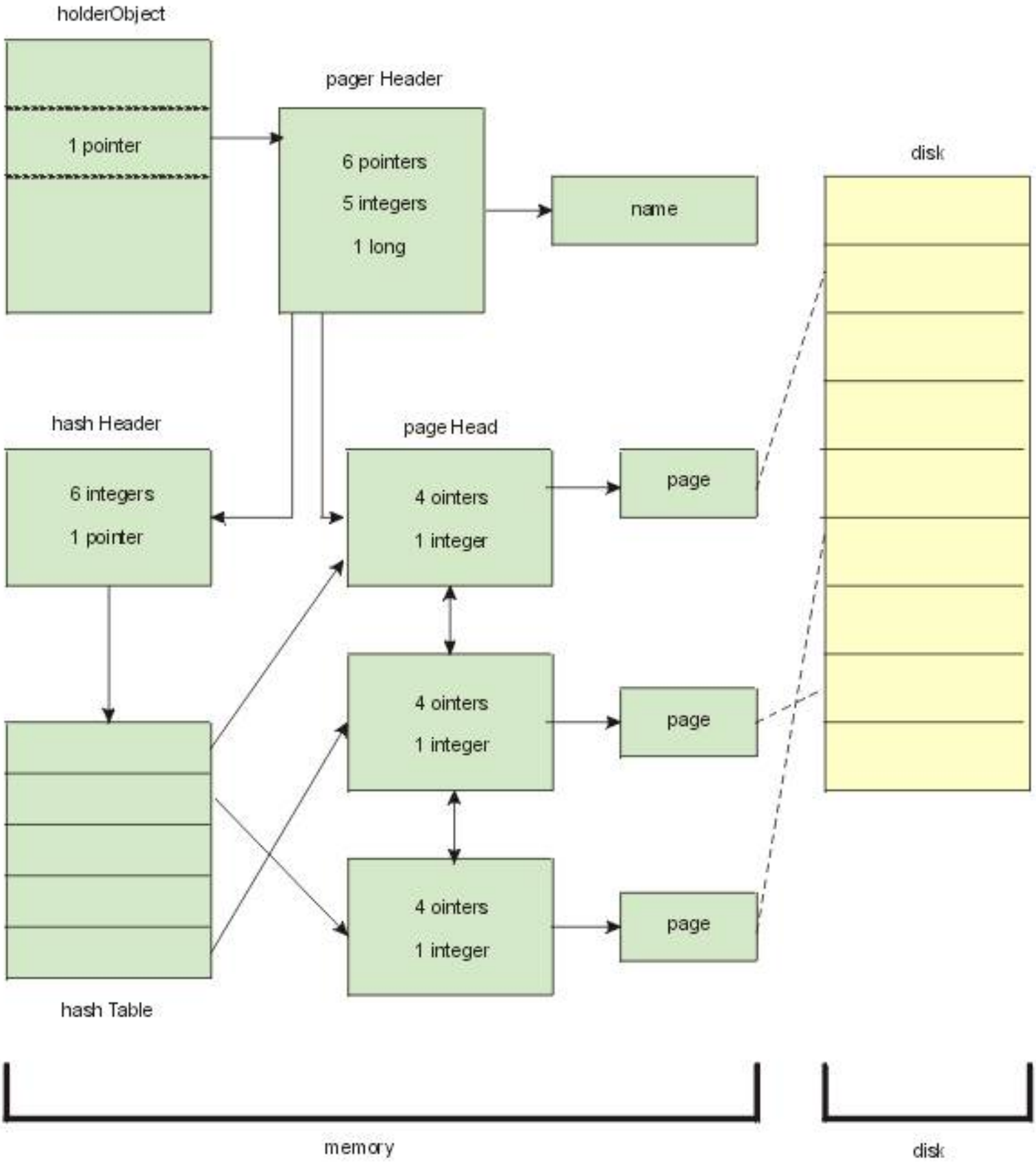
In many applications, the program needs more memory than is available in RAM. Virtual memory and extended memory systems help to overcome this problem by paging disk space into memory. Only the most often addressed pages (or pages addressed most recently) are kept in memory, which improves access speed, and creates an illusion that the application program works with an essentially unlimited size of memory.

Virtual or extended memory operates at the system level, and the application program cannot control its parameters, or open several independent memories at the same time. The PAGER organization described in this section allows you to create and control several pagers simultaneously. Each pager operates independently, with resources controlled by the application program. The disk/page memory is addressed by a *long int* index, and is treated as raw memory (a string of bytes). It can be used to store objects (structures), but the size of the objects, and any pointers must be managed manually.

There is no need to save pager data to disk. The information stored in a pager remains on disk even after the end of the run (each pager has its own disk file), and can be accessed later. The pager is automatically persistent.

For example, you can maintain two pagers running simultaneously, one having page size=10kB and up to 200 pages in memory, the other with page size=1MB, and up to 3 pages in memory. Each has its name (*id*), and you code with each pager as if it were a huge memory resident array, which you index by a *long int*.

Implementation



The figure above shows a slightly simplified internal organization of one pager. The figure does not show all the details of the hash table. It also shows the *holderObject* with only one pager. In reality, there is a linked list of pagers on each *holderObject*.

Inside the pager, each page has its own *pageHead*. Fast access to the pages is provided via hash table (exactly as described in Chap.11.13). Pages move automatically between disk and memory. The organization shown in the figure is completely transparent. When closing a pager, all memory resident pages move to disk, and the reserved memory is freed. When opening a pager, it automatically opens the old file if it exists, or else opens a new file.

ZZ_HYPER_PAGER(id,TYPE);	creates a pager on object type TYPE. *id* is the pager identifier.
*void form(TYPE *h,char *fileName,int pgSize,int numPgs,int init);*	forms a new pager (or re-opens an old one) on the *holderObject h. fileName* is the name of the disk file, *pgSize* is the size of one page in bytes, *numPgs* is the maximum number of pages in memory, and init specifies the initialization for new pages (0=none,1= `\0',2=` ').
*void io(TYPE *h,long ind,char *buff,int n,int mode);*	reads/write a block of data from the pager on object *h*. Here *ind* is the byte address within the file, *buff* is a memory buffer (target or source object of the read/write operation), *n* is the size of the object in bytes, and *mode* specifies whether reading from disk (=0) or writing to it (=1).
*char * addr(TYPE *h,long ind);*	makes sure that the page corresponding to disk address *ind* is loaded in memory, and returns pointer *ptr* to the *ind* location.
void close(hp);	closes pager with id=*id* on object *h* moves all pages to disk, and frees the memory.
long fill(hp);	returns the highest disk address currently used by the pager.
void flush(hp);	flushes all dirty pages to disk.

Limitations:

Under DOS or other systems with integers only 2 bytes long, the total internal memory per pager is limited to 32k (=32767). In case that the total of numPages*pageSize exceeds this limit when forming the pager,

numPages is automatically reduced to fit into the limit, and a warning is printed about it. The run continues uninterrupted; the only disadvantage is more intensive paging.

The reason for this limitation is that all the memory used by the pager is, internally, allocated as a single memory buffer. This design decision allows the pager to handle objects that are bigger than one page, or which overlap page boundaries.

Example:

test29c.c and *test29d.c*.

« Previous Section 11.13 Hash Tables Next Section 11.15 Access to Type Tables »

11.15 ACCESS TO TYPE TABLES

Purpose:

Internally, the C++ compiler keeps the information about the construction of classes (meta-class information), how classes inherit each other, where the virtual function pointers are, and about the offset of member objects within their parent class.

There are situations when knowing this information would greatly simplify the code. Sometimes, programs have to duplicate the analysis which has already been done by the compiler.

For example, if the meta-class information is readily available, you do not need an external class browser. You can traverse your classes directly in your program. You can also code your own graphical display of class relations.

Another situation when this information is useful is in the coding of a private version of persistent data. You must provide special treatment for inter-object pointers, and also distinguish between inter-object pointers and virtual function pointers (and virtual class pointers). These pointers can be in different parts of the object, depending on the particular compiler you are using and, except for the Microsoft 7.0 compiler, no compiler provides even the slightest clue to where these pointers are.

The TYPE class provides access to all these features, which are normally hidden features. The class does not access the tables that the compiler is using internally. It re-generates the information by analyzing the syntax of the classes that contain the *ZZ_EXT_*.. statement, and then applying a special algorithm (see The C++ Journal, two-part article by J.Soukup, starting in the fall issue of 1992), which is compiler independent.

Access is very fast (essentially just indexing into pre-calculated arrays); it does not perform any long searches.

Another useful service provided by this class is that, given a pointer to a base object, the class can find the pointer and type of the highest object derived from it.

Examples:

```
ZZ_HYPER_TYPE(type);
class Coord {
    ZZ_EXT_Coord
    int x,y;
    ...
};
class Shape {
    ZZ_EXT_Shape
    ...
};
class Tool {
```

```
    ZZ_EXT_Tool
    ...
};
class Image: public Tool, public Shape {
    ZZ_EXT_Image
    ...
};
Shape *sp;
int tp,tt,t1,t2,n;
void *vp,**vv;
char *cp;
zzTypeInfo inf;
Image *ip;
n=type.num(); // returns total number of types (here 3)
tp=ZZgetType(Shape); // gets type index
ip=new Image;
sp=(Shape *)ip; vp=(void *)sp;
type.trueType(&vp,&tp);
    // updates vp and tp to the true values (Image type)
type.info(tp,&inf); // returns info for type tp
printf("class=%s\n",inf.name);
type_iterator it(tp);
    // will iterate through base classes and members of tp
while(tt= ++it >>= 0){
    printf("sub-type=%d %s\n",tt,it.name());
    // prints: 'inherit','virtual','refer' or member name
// print all types and their subtypes/members
type_iterator it;
n=type.num();
for(t1=0;t1<n;t1++){ // go through all types
    type.info(t1,&inf);
    printf("\n%s= ",inf.name);
    it.start(t1);
    while(t2= ++it >= 0){
        type.info(t2,&inf);
        printf("%s:%s",it.name(),inf.name);
    }
}
type_pointers ptrs(sp,tp);
    // traverse HYPER pointers on sp
while(vv=ptrs++){
    printf("pointer at=%d leading to type=%d\n",vv,ptrs.nxtType()); }
// print virtual function pointers within an Image object
int i,tp,sz; void **p; char *p,*m; Image *ip;
zzTypeInfo inf;
```

```
ip=new Image;
tp=ZZgetType(Image);
type.info(tp,&inf);
m=inf.mask2;
sz=inf.size; // sizeof(Image) could be used instead
for(i=0, p=(char *sp); i<<sz; i++,p++,m++){
    if(*m='F')v=(virt **)(&p);
    printf("virt.fun.pointer at=%d value=%d\n",v,*v);
}
Structure that keeps the type info:
struct zzTypeInfo {
    char *name; // class name
    int size; // size of this class
    char *mask1; // 0-filled with correct v.f/v.c.ptrs
    char *mask2; // 'F' or 'C' for bytes
        // that start v.f./v.c. pointers
    int virt; // bin packed:
        // 01=abstr.class,02=virt.base class,04=v.f. present
};
```

Note that the two masks are needed when virtual function pointers have size, which is different from regular pointers. This happens, for example, when using Borland compilers with the large memory model.

For complete running examples, see *test39a.c*, and *test39b.c*.

Syntax:

```
ZZ_HYPER_TYPE(id)
int id.num(void); // returns total number of types
int id.vfCheck(void *ptr); // returns type index, <0 fails
int tp=ZZgetType(className); // macro gets type index
void id.virtRange(void**,void**); // returns the range
    // of all virt.functions
void id.trueType(void**,int*); // given pointer and type
    // of a base object, updates them to the top level
zzTypeInfo* id.info(int); // gets type info
id_iterator it; // declares general iterator
id_iterator it(int); // initialized for the given type
void it.start(int); // start for the given type
char * it.name(void); // returns:'inherit','virtual',
    // 'refer', or the member name

int t; // traverse sub-types
id_iterator(tp);
while(t= ++it){ ...
```

```
void *v; // traverse HYPER pointers
id_pointers pt(void*,int);
while(v= ++it){...
```

The purpose of function *int vfCheck(void *ptr)* is to detect type for unknown (untyped objects). This situation is described in more detail in Chap.11.10.

Previous Section 11.14 Pager

Chapter 12: Documentation

12. DOCUMENTATION

12.1 On Line Help

OrgC++ has an intelligent help function, which allows you to query about the organizations and functions that are in the library. This function automatically includes organizations/features added by the user.

On-line help uses a free format, and has no fixed syntax or menu. You simply enter, from the main OrgC directory:

zzhelp <string>

The program lists the OrgC++ features that best match the given string, and allows you to select commands for which you would like to see more documentation. The program retrieves the selected documentation, and places it into the file *zzinfo*, which you can browse using a standard editor or the UNIX *more* utility.

The query must be presented as a single string, even if it combines several words. For example: ADDRING, add_RING, and ringADD, will all produce the same result.

Each line on the screen shows: long name, short name, and the file in which the information is stored.

Under UNIX, the file *zzinfo* is automatically *nroff-ed*. Under DOS, the file has the raw *nroff* format, with typesetting instructions in it (*format-me*).

Using on-line documentation for the C++ version of the library requires a slightly different approach. You always start with the file that contains the *ZZ_HYPER_* declaration. Let us assume, for example, that you are looking for details on how to delete an object from the *DOUBLE_TREE*. You proceed in these steps:

Example: *zzhelp ADD_RING*

Best match found:

[0] macro:	*ZZ_ADD_SINGLE_RING*	ZZadd1R	file: addsring
[1] macro:	*ZZ_ADD_DOUBLE_RING*	ZZadd2R	file: adddring

select numbers:

Example *zzhelp ringADDS*

Best match found:

[0] macro:	*ZZ_ADD_SINGLE_RING*	ZZadd1R	file: addsring
[1] macro:	*ZZ_ADD_DOUBLE_RING*	ZZadd2R	file: adddring

select numbers:

Example*: zzhelp 1_TO_N*

Best match found:
organization=1_TO_N

[0] macro:	*ZZ_ORG_1_TO_N*	ZZorg1toN	file: org1ton
[1] macro:	*ZZ_TARGET_1_TO_N*	ZZtarg1toN	file: targ1ton
[2] macro:	*ZZ_SOURCE_1_TO_N*	ZZsour1toN	file: sour1ton
[3] macro:	*ZZ_FORWARD_1_TO_N*	ZZfwd1toN	file: fwd1ton
[4] macro:	*ZZ_BACKWARD_1_TO_N*	ZZbwd1toN	file: bwd1ton
[5] macro:	*ZZ_A_TRAVERSE_1_TO_N*	ZZAtrav1toN	file: trav1ton
[6] macro:	*ZZ_A_RETRACE_1_TO_N*	ZZAretr1toN	file: retr1ton
[7] macro:	*ZZ_ADD_RELATION_1_TO_N*	ZZadd1toN	file: add1ton
[8] macro:	*ZZ_DELETE_RELATION_1_TO_N*	ZZdel1toN	file: del1ton
[9] function	*ZZF_ADD_RELATION_1_TO_N*	ZZFadd1toN	file: fadd1ton
[10] function:	*ZZF_DELETE_RELATION_1_TO_N*	ZZFdel1toN	file: fdel1ton

select numbers: 0 7 9
view file zzinfo ...

Example: *zhelp NewYorkCity*
nothing meaningful found

Find out which file contains *ZZ_HYPER_DOUBLE_TREE*. That can be done in two ways:

Method A:

cd orgC
zzhelp DOUBLE_TREE

This will give you a list of all the commands for this organization, including the file name in the most right hand column. In this case, the name will be *hypdtree.*

Method B:

UNIX	DOS
cd orgC/macro	*cd orgC\macro*
is hyp*	dir hyp*

You will get a list of about 30 files with names that are a bit cryptic but still meaningful. You will see that hypdtree clearly resembles the organization you are looking for.

The next step is to view (using your favourite editor) the file *orgC/macro/hypdtree*. The file contains the declaration of the interface class, iterator, and all the methods. You will see that the function *del()* is indeed there, and how it must be called. The big advantage of C++ is that all things related to one object are neatly kept together.

When you are looking into this file, you will also see that the actual code that deletes the object from the tree is hidden under a macro, in this case *ZZ_DELETE_DOUBLE_TREE()*. This has two reasons: (a) it makes the C and C++ code perform identically, (b) it saves a lot of coding.

If you followed Method A above, you already know the name of the file, where this macro is located (*deldtree*). You could easily have guessed it (the naming of all files follows the same convention; the only restriction is that they must be not more than 8 characters in order to make the whole system compatible with DOS). Another possibility is to call the interactive help again:

zzhelp DELETE_DOUBLE_TREE

The last step is to view (using your editor again) the file *deldtree*. This file contains all the technical information available about the algorithm, its behaviour, possible errors, etc. Each file also contains a short example which is, at the present time, mostly in C. We are gradually adding C++ related documentation to all of the files, but this is a long term process.

Note that there is a direct mapping of calling parameters between C macros and C++ methods. You can either look in the tables shown in Appendix A, or remember these rules:

- If the call does not return any values, parameters are the same, except for the organization *id*

which, in C++, is used differently: *ZZ_DELETE(id,par,obj)*; translates into *void id.del(par,obj)*; *ZZ_SAVE(file,n,v,t)*; translates into *util.save(file,n,v,t);* because there is no UTILITY class in C (save and allocation utilities are simply global).

- If the call returns a parameter (and it never returns more than one, usually the last one), the parameter moves ahead of the function: *ZZ_CHILD(id,par,ch)*; translates into *ch=id.child(par);* *ZZ_FORWARD(id,obj,next)*; translates into *next=id.fwd(obj);*

If you find this system a little bit too complex for the beginning, remember that you don't have to use it until you start to add or modify the library. The printed Users' Guide contains all the information you need to use the libraries.

12.2 Reference Guide

OrgC++ documentation consists of three documents:

- User's Guide;
- Reference Guide (this chapter describes details);
- On-line help (explained in Chap.12.1).

The User's Guide is shipped with the software, and is available in printed form only.

You can generate the Reference Guide automatically from within the *orgC/docum* directory, by typing: *zzdocum <mFile>*

If you don't specify any mFile, the program creates a complete Reference Guide, if you give an *mFile*, it generates only an update for commands/macros specified in the file. For C++, provide *ZZ_HYPER_*declaration names, plus corresponding C-command names (see Appendix A). In either case, the generated document includes an index by organization/function name, and one UNIX-like page for each function or macro. Documentation for every feature includes an example, and is identical to the information you obtain when calling *zzhelp*.

The text of the Reference Guide will be stored in the file *orgC/docum/ZZrefer* and will be in *nroff -me* format. If you run under DOS or do not have *nroff -me* available on your UNIX system, you can call *orgC/zzroff*, which will format the text for the printer. Assuming that you are in the *orgC/docum* directory, you have to type:

```
..\zzroff ZZrefer temp
print temp
```

Note that OrgC is using an advanced concept of creating documentation directly from the source code. OrgC macros and functions are stored in the *orgC/macro* directory, with the source code and the documentation together in each file. The documentation program retrieves the Reference Guide directly from these files, while the *zzcomb* program (see Chap.16), retrieves the source code stripped of

comments and generates *orgC/zzcomb.h.*

If you require only the Reference Guide update, list only those macros/functions for which you want documentation.

Example of *mFile:*

> *ZZ_ADD*
> *ZZ_DELETE_SINGLE_TREE*
> *ZZpar1T*

WARNING: *A zzdocum* run always overwrites the old file *ZZrefer*. If you want to generate update pages only, rename *ZZrefer* before calling *zzdocum.*

« Chapter 11: Available Organizations Chapter 13: Memory Management »

13. MEMORY MANAGEMENT AND SAVING ON DISK

13.1 VIRTUAL FUNCTIONS AND INTERNAL POINTERS

Familiarity with this section is not essential for use of the library. It only provides more depth and theoretical background.

If you are just starting, and you want to use OrgC++ quickly, skip this chapter and proceed directly to Chap. 13.2.

When you are saving objects in C++, the program must correctly save and retrieve three parts of the object:

1. General attributes, *int* and *float* types.
2. Internal pointers that form the organization of data.
3. Virtual function pointers, and virtual base class pointers.

All this has to be done recursively for all member objects of the given class, and following the class hierarchy.

We will not explain in full detail how this is done, only the overall concept. If you understand the underlying organization, you will appreciate more what OrgC++ can do for you; to implement persistent data is indeed a complex task.

Let us look at this example:

```
class Obj2 { // base object
    ZZ_EXT_Obj2
public:
    virtual prt(void){ };
    virtual save(Obj2 *p){ };
};
class Obj1 : public Obj2 {
    int i;
    ZZ_EXT_Obj1
public:
```

```
    Obj4 m; // member object, not pointer
    virtual prt(void){cout <<<< i;};
    virtual save(Obj2 *p){ .... };
};
class Obj3 { // another base object
    ZZ_EXT_Obj3
public:
    virtual abc(int i){ };
};
class Obj4 : public Obj3 {
    ZZ_EXT_Obj4
    int k;
    char *p;
public:
    virtual abc(int i){ ... );
};
class Obj5 { // simple, not interesting
    ZZ_EXT_Obj5
    ...
};
ZZ_HYPER_DOUBLE_RING(myRing,Obj2);
ZZ_HYPER_SINGLE_TRIANGLE(myTria,Obj4,Obj5);
```

A typical internal representation of one Obj1 may be like this (depending on the C++ compiler):

fwd	ring pointer (ZZ_EXT_Obj2)
bwd	ring pointer (ZZ_EXT_Obj2)
vf	virt.function pointer Obj2
i	integer in Obj1
vf	virt.function pointer Obj3
child	pointer (ZZ_EXT_Obj4)
k	integer in Obj4
p	character pointer in Obj4

When virtual base classes are used, one more internal pointer is present, subject to different rules than the pointers shown here.

When restored from disk, objects are in new memory locations, therefore pointers *fwd*, *bwd*, and *child* must be modified to point to the new objects. If you store the object during one program run, and restore

it in another run, the virtual function tables will also be in a different location. The remaining attributes should keep their values (*i*,*k*), with perhaps the exception of some attributes, such as *p* here, which is only a temporary variable, and does not require permanent storage.

Note several interesting details: There is only one virtual function pointer in Obj2, even though two virtual functions are present. Some objects (Obj1,Obj4) do not have their own virtual function pointer, even though they are using virtual functions.

The value of the virtual pointer is the same for all objects of the same type, however, it is different, for example, for a standalone Obj2, and for Obj2 which is a subclass of Obj1. Most traditional C++ compilers like AT&T C++ place the virtual function pointer at the end of the class. However, the Microsoft 7.0 compiler places it at the beginning of the class.

Inheritance inserts the sub-class at the beginning of the object. Pointers which form the data organization, and which are hidden under *ZZ_EXT* statements, are dispersed throughout the object.

In order to manage the whole situation, OrgC++ must know:

1. the inheritance/member hierarchy (*ZZ_EXT* statements and class declarations provide this information);
2. which classes have virtual functions;
3. where the pointers that need an update are (*ZZ_EXT* statements hide them).

When you create a new object, data pointers must be initialized to NULL (disconnected pointers), while virtual function pointers are set to the same values for each given class. In order to initialize the pointers, older versions of OrgC++ required the *ZZ_INIT() macro* to be present in all constructors.This is no longer necessary: pointers are initialized automatically, except for special situations like with the use of *SELF_ID*.

When using an allocation algorithm which generates a raw memory block, the block must be initialized in order to become a valid object. The best way to do that is to create one fully initialized instance of each class, and copy its contents into the new block. When saving to disk, objects are saved in their original form (with the old pointers). For every object, there are two basic pieces of information: a fix-sized header which provides the old location of the object, its type and size (the object can be an array); and then the object itself, sometimes in several records.

When retrieving the information from disk, the program builds an internal table of old/new addresses for all objects, and uses it to convert data pointers from old to new locations.

The handling of binary and ASCII data is conceptually different:

Binary data is loaded from disk as a block of bytes. That is very fast, but the virtual function pointers have to be updated. On the other hand, ASCII data can start with a pre-initialized class instance which

has virtual function pointers already in place; the *open()* function then only inserts data pointers and the remaining attributes.

In both cases, data pointers must be converted to their new locations. This, again, is a recursive procedure if the object is composed of other objects.

Chapter 12: Documentation

Next Section 13.2 Memory Management

13.2 MEMORY MANAGEMENT (ALLOCATION/ FREE)

In addition to static checking through strict typing, OrgC++ provides run-time protection against dangling pointers. The idea is simple:

When a new object is allocated, all automatically managed pointers (those which hide under *ZZ_EXT*_statements and are transparent to the user) are initialized to NULL. An object can join a new organization or be de-allocated only when all of its pointers are NULL. The *del()* function disconnects the object, and sets all its pointers to NULL.

Protection against dangling pointers is complete for those organizations where objects are mutually connected or, in other words, where each object within the organization is connected by at least one pointer to its remaining objects. Most organizations such as *RING*, *TREE*, *GRAPH*, or *DOUBLE_LINK* are of this kind, even if they are singly linked. However, *SINGLE_LINK*, *GENERAL_LINK*, and *NAME* are exceptions; their targets carry no information on whether they have been disconnected.

DOL is not protected against the allocation of an array of objects and then attempting to free each object separately. For this reason, we strongly recommend always allocating one object at a time, and applying the *ARRAY* organization only when dealing with index-accessed objects.

The automatic *clear()* function, which deallocates the entire data organization, works only if all objects were allocated individually. However, an automatic *save()* saves the whole organization on disk, regardless of how the objects were allocated; even automatically allocated objects may be present. For more on these functions, see Chap.13.3.

Important:

When you allocate an object or an array of objects from the heap (using operator *new*), DOL initializes all data organizations as empty, with all the internal data structure pointers set to NULL. For example:

```
class Town {
    ZZ_EXT_Town
    ...
};

void foo(...){
    Town *myTown;      // pointer, not object
    myTown = new Town; // internal pointers automatically initialized
```

This automatic initialization is not provided for automatically allocated objects, is for example:

```
class Town {
    ZZ_EXT_Town
    ...
};

void foo(...){
    Town myTown; // object, internal pointers not initialized
```

When using DOL, you can use one of the following two strategies. (a) Not to worry about this issue and, consistently, allocate all objects from the heap. (b) Insert *ZZ_INIT* either into constructors of all your classes that have ZZ_EXT_...

(prefered strategy), or at least into those classes that have automatically allocated objects:

```
class Town {
    ZZ_EXT_Town
    ...
public:
    Town(){ZZ_INIT(Town); ... your code ...}
    Town(int i){ZZ_INIT(Town); ... }
    ...
};

void foo(...){
    Town myTown; // object initialized as disconnected
```

Since DOL implements all data structures as pointer-linked rings, it allows powerful and yet inexpensive integrity checking. For example, any time you *add()* an object to any data structure, DOL checks that all its internal pointer(s) are NULL. It is impossible to mess up the data structure or create stray pointers, and it all happens transparently.

One of the typical ways to crash any program is to destroy an object without disconnecting it from a linked list or other data structure. All data structures in DOL provide *del()* function which disconnects a given object. But what if we forget to call *del()* before destroying the object?

If you insert *ZZ_CHECK(objType)* into the destructor(s) of a class which has the *ZZ_EXT* statement, any time you try to destroy an object of that class, regardless whether it was allocated automatically or from the heap, all its internal pointers are checked for being NULL. If any pointer is not NULL, it means the object was not properly disconnected, and a detailed warning message is issued which tells you which data structure caused the problem. Unfortunately, the C++ language does not allow to interrupt the destruction process once it started, so after printing the message your the program will likely crash or mulfunction. This will not be a blind crash though. You will know exactly where the problem is, and will be able to correct it without any debugging.

```
class Town {
    ZZ_EXT_Town
    ...
public:
    Town(){ZZ_INIT(Town); ... your code ...}
    ~Town(){ ... your code ...; ZZ_CHECK(Town);}
    ...
};

void foo(...){
    Town myTown, *townPtr;
    ...
    delete townPtr; // checks whether this is safe
}   // on return from foo(), myTown is checked before destruction
```

For examples of the use of *ZZ_INIT()*, see *test4d.c, test27a.c, test29d,* and *test31.c.* In *test4d.c*, the use of *ZZ_INIT()* is essential, at least for class *Apple*. In *test29d.c*, the macro is there for historical reasons, but could be removed without any effect on the operation of the program. For examples of *ZZ_CHECK()*, see *test4d.c, test0n.c* and *test31.c.* Note that *ZZ_CHECK_FREE(Pin,1,this)* is an older equivalent of *ZZ_CHECK(Pin)*. For another example, see the end of Chap.6.

For objects that do not have any hidden pointers (their classes are not used in any of the *ZZ_HYPER_*.. statements), neither *ZZ_INIT()* nor *ZZ_CHECK()* should be used.

Using the internal free lists:

Some programs have to deal with a large number of objects that are frequently allocated and freed. The standard allocation available on most systems may lead to memory fragmentation, causing performance deterioration, and possibly program failure due to insufficient memory. In particular, this is very important for memory blasting (see Chap.13.3.), which does not automatically reuse the space from discarded objects. Operator *delete* there is overloaded to do nothing, and if we don't do anything else, each destructed object would leave an unused hole in the memory space. Since memory blasting saves the entire memory image to disk, this would also dramatically increase the storage on disk.

In such situations, the best strategy is to keep a list of free objects, and reuse them instead of allocating them again.

The library has a manager of the free space, which hides under *ZZ_HYPER_UTILITIES(util)*. This manager works with three types of objects:

1. registered classes (those that have *ZZ_EXT_*..);
2. other objects and text strings that do not exceed in size the largest registered object (*maxSz*);
3. large objects and arrays.

The manager keeps a set of free lists, one for each object size up to *maxSz*, with the increment of 4 bytes. Free objects are linked by a pointer which overlays the first four bytes of the object. No additional memory is needed in order to maintain the free list. There is one linked list of large objects, with each object keeping pointer to the next object, and its own size.

There are three pairs of functions to allocate/free objects in this manner. Mixing these calls can lead to serious errors.

delete, delete[]	for registered objects
util.newStr() *util.delStr()*	NULL ending text strings
util.newArr() *util.delArr()*	plain blocks of memory

Note how different cases are handled internally. Functions *new()* and *delete()* know the class and its size, and can reuse the the space without a header or any other overhead. Functions *newStr()* and *delStr()* rely on the \0 character to determine the size of the string; they also do not use any overhead for the memory management. Functions *newArr()* and *delArr()* keep the size of the space internally (overhead of one *int*). The reason for this is that these operations are typelss, and *delArr()* must keep the size of the object.

Examples of correct reuse:

```
class Town { ... }; Town* tp=new Town; delete tp; //registered class
char* s=util.newStr("John Brown"); util.delStr(s); //fixed string reuse
int i[]; int n=12000;
i=(int*)util.newArr(n*sizeof(int)); util.delArr(i); //general block
char* s=(char*)util.newArr(80); strcpy(s,"Joe Doe");
util.delArr(i); // text buffer
```

Examples of incorrect use:

```
// no possibility of problems with new() and delete()
char* s=util.newStr("John Brown"); delete(s); // will crash
char* s=new char[12]; strcpy(s,"John Brown");
util.delStr(s); // will crash in open
char* s=util.newStr("John Brown");
strcpy(s,"Joe Doe"); util.delStr(s); // memory leak
char* s=util.newStr("John Brown"); util.delArr(s); //will crash
```

Control of free storage:

Function *util.freeCount(int sz)* returns the present number of free objects of this size. Using *sz=0* returns number of large (variable sized) objects.

You can turn on the free store by calling *util.useFreeStore(1*), and turn it off by *util.useFreeStore(0)*. For memory blasting,the default is free storage on.

Instead of globally invoking free lists for all registered classes, you can invoke free lists only for selected classes. ZZ_OBJECT_NEW(type,ptr) is a macro which picks up the next object from the free list (or allocates it if the free list is empty); ZZ_OBJECT_FREE(type,ptr) places the object on the internal free list.

You can control (and mix) the regular allocation with free lists:

```
class myObj {
    ZZ_EXT_myObj
publ:
    static myObj *newObj(){ myObj *p; ZZ_OBJECT_NEW(myObj,p); return(p); }
    void delObj(){ ZZ_OBJECT_FREE(myObj,this); }
};


myObj* obj1 = new myObj; // straight allocation, not using free list
myObj* obj2 = myObj::newObj(); // using free list
delete obj1; // destroys object regardless of where it came from
obj2->delObj(); // move obj2 to the free list
```

util.clear(void) deallocates all internal free lists.

For an example of how to use the free list allocation, see *test23b.c*

Handling names:

In addition to free lists of text strings (see above), DOL also helps with the general management of names. It has the organization *ZZ_HYPER_NAME* (Chap.11.5 - similar to a String class), which treats names as instances of a string object, and it provides additional functions that facilitate creating/freeing names:

*char *ptr=util.strAlloc(char *name)* for a given *name* allocates the appropriate space, copies the name into it, and returns a pointer to it.

*void util.strFree(char *ptr)* frees the given string.

IMPORTANT: When saving data to disk, all names must first be allocated with *util.strAlloc()*, and then registered as *NAME*, if you use automatically allocated strings like "abcd" (for more details, see page 11.6.2).

For examples of how to handle variable length names, see *test0n.c*, *test9a.c*, *test16c.c*, and *test25b.c*.

Allocating arrays:

DOL also provides a package for the management of arrays (see Chap.11.12). Note that binary heaps and hash tables, which are both based on arrays, use the same method of allocation/deallocation as arrays do.

In DOL, an array exists as a data organization attached to an object.

ZZ_HYPER_ARRAY(id,HOLDER,TYPE); declares that every object of type HOLDER can have an array of objects type *TYPE*.

*TYPE *arr=id.form(HOLDER *obj,int size,int increment);* this function allocates an array of size objects, initializes the pointers of all its members, and properly sets the internal array header. It also sets the size *increment*, and returns the pointer to the array (*arr*) for fast unprotected indexing. It is recommended not(!) to use *arr* directly (as in *arr[i]*, but through the index function, *id.ind(obj,i)*. The array may reallocate itself, and render the reference invalid.

*void id.reset(HOLDER *obj,int size,int increment)* resets the control parameters of the array (current *waterMark* and *increment*), without re-allocating the array or changing its actual content.

*void id.free(HOLDER *obj)* frees (destroys) the array.

For an example of allocating/deallocating arrays, see *test16c.c*.

Private allocation schemes:

If you have a private memory management system, which you prefer to use, allocate objects through your system, and then initialize them with *ZZ_INIT()*. It is even better to convert the entire DOL to your memory management system:

File *orgc/lib/msgs.c* contains functions *ZZmassAlloc()* and *ZZmassFree()*, which control allocation/deallocation throughout DOL. Replace *malloc()* and *calloc()* there by your private functions, and recompile the library.

Big block of memory

One of the methods available for fast storage to disk (persistency) is based on the idea of reserving a block of memory for allocation of runtime objects, and restoring them from disk into the same relative position in another memory block, which has the same size. The advantage of this arrangement is that mutual offsets between objects remain constant, which allows fast updating of pointers after restoring the data from disk.

The disadvantage of this arrangement is the need to estimate the size of required memory, which is often difficult to predict. Also, even though the memory is reserved as one compact block, the data is saved to disk object-by-object, and even when buffered, it cannot match the efficiency of *memory blasting* .

The big block allocation is still in DOL mainly for historical reasons. Some large commercial applications use this option, and do not want it to disappear from the system. Two functions control big block allocation:

void util.blkAlloc(int n,0) allocates a new block of *n* bytes. If called several times, the most recent block acts as the currently active block.

void util.blkFree(int i) controls the use of the existing block. i=0 deallocates the active block; i=1 keeps it but, temporarily, returns to the normal allocation algorithm; i=2 turns allocation back to the big memory block.

There are no special *SAVE/OPEN* commands for this option - regular *util.save(...)* and *util.open(...)* are used. If the data was saved as one big block of memory, *util.open(...)* makes the following attempts:

1. If there is no active memory block, it opens a block of the same size as was originally used, including the free section of the pool.
2. If there is an active memory block that can contain the new data, the block is used, and the free pool is appropriately reduced.
3. If there is an active block of memory, but its free area is not large enough, the program creates a new block, which becomes a new active block.
4. If all allocation attempts fail (block size is too big), the program issues a warning message, and returns to normal allocation mode.

When the active pool is exhausted, a warning message is issued, and the operation automatically switches to normal allocation mode. Your program can safely continue the run. You can even save your data to disk. All you lose is the speed when re-opening the data from disk.

Allocation based on internal free lists works together with the memory block.

Note that all internal data, such as stacks and tables used by *util.save()* and *util.open()* are allocated from this the same memory block. (This is not true for other memory allocation modes.)

The current memory assigning algorithm is simple. For every new object, the high water mark of the given pool is raised. Unless you use the free-list option, discarded objects are not reused.

If *util.blkAlloc()* is called several times, the old data remains in memory. The most recently allocated block becomes the active block.

util.blkFree(1) allows you to separate the allocation of useful objects from temporary data, as in the following example, where all the application data has been allocated through the memory block, but then before calling *util.save()* allocation switches to normal mode, thus avoiding the allocation of temporary data, internal only to *util.save()* from the same block.

```
ZZ_HYPER_UTILITIES(util);
myObj *p,*s,m;
util.blkAlloc(3500,0); // reserves block of 3500 bytes
...
p=m.newObj(1); // create object
s=m.newObj(1); // create object
...
util.blkFree(1);// switch to normal allocation mode
util.save(...); // normal allocation, inter. tables
util.blkFree(0); // free the memory block
```

Free lists:

When allocating objects from a block of pages, you can use either *new()* or *ZZ_OBJECT_NEW()* which also provides, on top of memory paging, a list of free objects by type.

Use *new()* and constructors as usual, but **never** place *ZZ_OBJECT_NEW()* into a constructor. Look at *test23b.c* for the

correct way to use *ZZ_OBJECT_NEW()*.

Error conditions:

If a block of a given size cannot be allocated, *util.error()* returns 01, and a warning message is printed. The situation is the same as when exhausting the available pool. You can continue to run, but *util.open()* will allocate memory for individual objects, not from one big block.

Examples:

DOL used to be available both for C and C++, and the test suite does not have any examples of the big block allocation in C++. For examples in C, look at *test0k.c*, *test24c.c* and *test33b.c*. Note that *ZZ_BLOCK_ALLOC()* and *ZZ_BLOCK_FREE()* are identical to *util.blkAlloc()* and *util.blkFree()*.

« Previous Section 13.1 Virtual Functions and Internal Pointers

Next Section 13.3 Saving on Disk »

13.3 SAVE/OPEN

The prime purpose of the *save()* and *open()* is, in a single command, to save and retrieve the entire data space to/from the disk. This includes all the objects and the relations they form. Later on, we will discuss advanced features such as saving individual objects, saving data in several pieces, saving the entire data space and updating a few objects later, plus the version control of both your program and of DOL. Let's start with the basic use of *save()* and *open()*.

In practically any program, objects and their relations form a network, which connects all the objects together. Also, there is usually one or a few root objects -- we call them 'key entries' -- which have the property that if we start from them and follow the data structure links, we will reach all the objects the program is using. This does not mean that we can reach all the objects from any key entry; rather any object can be reached from at least one key entry.

Sometimes, there are a few additional objects which play some central or key role in the data structures, and you may want to know how to access these important objects after you download the data from the disk. In that case, include also these objects as 'key entries'.

When you want to save your data to disk, you call *void save()* and give it the name of the file where you want the data to store, number of key entries, key entries as (void*) pointers, and names of the classes of the key entries. For example:

```
class A {
    ZZ_EXT_A
    ...
};
class B {
    ZZ_EXT_B
    ...
};
class C {
    ZZ_EXT_C
    ...
};
... many other classes ...
... many ZZ_HYPER statements ...
ZZ_HYPER_UTILITIES(util);

A *a; B *b; C *c; ...
void *v[2]; char *t[2];    // two key entries a,b

...
v[0]=a; t[0]="A";
v[1]=b; t[1]="B";
util.save("myFile",2,v,t); // saves the entire data space to the disk
```

Function *open()* uses the same syntax, but in reverse: It moves the data from the given file to memory, updates all the pointers, and returns the given number of key entries. Naturally, the number of key entries must be the same when saving and retrieving the data.

```
util.open("myFile",2,v,t); // retrieves the entire data space from the disk
a=(A*)(v[0]); // first key entry, must cast
b=(B*)(v[1]); // second key entry, must cast
```

Function *clear()* as similar parameters as *save()* and *open()*, except that no file name is given. This function traverses the

memory data and destroys it. There are also two ways to check that the save/open operations worked correctly. If *util.error()* returns a non-zero value, and error occured. Also, after *open()*, no returned key entry should be NULL, and the type names should agree with those used when saving the data. Here a sample of the code which demonstrates these features, this time using only a single key entry. Note that *save()* and *open()* can be used multiple times, using the same or different file names:

```
class A {
    ZZ_EXT_A
    ...
};
... other classes
... ZZ_HYPER statements
ZZ_HYPER_UTILITIES(util);

A *a1, *a2, *a3; void *v[1]; char *t[1];

....
v[0]=a1; t[0]="A";
util.save("file1",1,v,t);
if(util.error())... // error recovery

util.clear(1,v,t); // destroy all the data in memory
...

v[0]=NULL; // good for error detection
util.open("file1",,v,t); // retrieves the entire data space from the disk
if(util.error())... // error recovery
if(!(v[0]) || !(t[0]) || strcmp("A",t[0]))... another check for correctness
a2=(A*)(v[0]); // key entry for the new data

// Note: If clear() were not used, we would now have two copies of the entire
//       data space in memory, one starting from a1, one from a2
// Note: clear() works only for the memory blasting mode - see the
//       "List of functions and their syntax" at the end of this Chapter.

... calculation which changes the data
v[0]=a2; t[0]="A";
util.save("file2",1,v,t); // saving into a different file
if(util.error())... // error recovery

...
if(someProblem){  // return to the original data from file1
    util.open("file1",,v,t); // retrieves the entire data space from the disk
    a3=(A*)(v[0]); // key entry for the new data
    if(util.error())... // error recovery

    v[0]=a2; t[0]="A";
    util.clear(1,v,t); // destroy the old data space

    a2=a3; // replace it by the one from file1
}
```

Important: If you plan to save/retrieve your organization from disk, it is imperative that you use the *NAME* organization for text strings. *NAME* is the DOL equivalent of the String class. Without using *NAME*, text strings will not be properly retrieved from the disk. Examples:

```
class A {
    char *tName; // general advice: avoid using this
    ZZ_EXT_A
public:
    void prt();
};

ZZ_HYPER_NAME(aName,A); // each A object is associated with string aName

void A::prt(){
    cout << tName << "\n"; // print the temporary, non-persistent name
    cout << fwd.aName(this) << "\n"; // print the persistent name
}
```

Other things to remember:

1. Organizations *GENERAL_LINK* and *SINGLE_LINK* will cause problems in *save()* if the target class does not have *ZZ_EXT*. This shows up as a run-time error. The DOL code generator (*zzprep*) cannot check for this condition.
2. If you use arrays, avoid referencing into an array by pointer. That includes, for example, an array of objects which also form a linked list. Even though DOL can handle such situations (the array must not re-allocated itself - must have declared as having a fixed size), this is a potentially dangerous practice that should be avoided. Try to always reference arrays by indices.
3. If you plan to remove all the data by calling *clear()*, do not use automatically allocated objects. *clear()* will include such objects in the set of all objects, and will attempt to deallocate them, which will result in a crash.
4. When *ZZ_INHERIT* is defined in the environ.h file, which happens in most C++ programs, the function *util.clear()* works only in the memory blasting mode. It is internally disabled (bypassed) in the other two modes - see the syntax section.
5. Memory blasting works only in environments where *sizeof(char*)==sizeof(int)*.

Examples:

```
class Apple {
    ZZ_EXT_Apple
public:
    int cIndex;
    Cherry *cPtr;
    char colour;
};
class Plum {
public:
    float size;
};
class Cherry {
    ZZ_EXT_Cherry
public:
    int weight;
};
class Holder {
    ZZ_EXT_Holder
};
ZZ_HYPER_ARRAY(arr,Holder,Cherry);
ZZ_HYPER_GENERAL_LINK(gLink,Apple);
ZZ_HYPER_SINGLE_LINK(cLink,Apple,Cherry);
```

```
Apple a,*aa;
Plum p,*pp;
Cherry c,*cc;
Holder h,*hh;
// CASE (1)
gLink.add(&a,&p); // is correct, but a and p cannot be saved unless ZZ_EXT_Plum is
used
// CASE (2)
cc=arr.ind(&h,8);
a.cIndex=8; // correct reference into array
cLink.add(&a,cc); // incorrect reference into array
a.cPtr=cc; // incorrect reference outside OrgC
cLink.add(&a,&c); // correct, not into array
a.cPtr=(&c); // correct, but save will not include this
// CASE (3)
hh=new Holder;
// util.save() can save both &h and hh, but
// util.clear() can be called only on hh
```

Note that when saving in binary format (none of these examples saves in only(!) binary), *ZZ_FORMAT()* statements are not required.

Three formats of saving data to disk

DOL provides three optional formats for saving data to disk:

1. The *binary format* is the default. It is the easiest one to incorporate into your program. It runs faster and takes less disk space than the ASCII format.
2. The *ASCII format* is the only format portable across different platforms and compilers. It also supports version control and schema migration. For example, you can have the same program running on a Sun workstation (under UNIX), and on an IBM PC (under WinNT). If you *save()* the program data on the Sun, and *open()* the file on the PC.
3. *Memory blasting* far outperforms the other two methods, but it is trickier to use. Also, destroyed object are not removed from the memory, so it may be necessary to combine this method with the use of built-in free lists.

Examples:

Almost every test program in the *orgc/test* directory demonstrates saving to disk.

1. *test0n.c* shows netlist processing with ASCII saving, *test0m.c* is another variation of the same problem with both binary and ASCII saving in the same program.
2. *test23a.c* demonstrates the saving of objects which contain virtual functions.
3. *test16c.c* saves arrays, selfID, timeStamp, and properties.
4. *test23f* demonstrates saving with memory blasting.

Method 1: Saving in binary format

If you don't invoke any special features and use *save()* and *open()* just as we showed above, the storage is performed in the default mode, using a 'binary format'. This means that the data space is stored in a binary form, as opposite to the ASCII - human readable format.

When you call the DOL code generator *zzprep* prior to the compilation, *zzprep* analyzes all the class definitions marked with ZZ_EXT (this is equivalent to a partial compilation), and generates the serialization functions. Note that this happens automatically, and it guarantees a match between the output and input serialization function. If you have used a system where serialization functions have to be manually coded, you know that it is a lot of work, and the result is highly error prone.

zzprep stores the definitions of the serialization functions under *ZZ_EXT_*.. statements, and their implementation in one of the files it generates, *zzfunc.c*. Also, under the *ZZ_EXT_*.. statements, *zzprep* stores all the pointers which form the data structures. Since DOL knows where these pointers are, the *save()* command can traverse all objects, starting from the key entries, and following these pointers from object to object.

The disk file starts with a header, which describes details of how all your classes are constructed: Their sizes, inheritance, member objects and their types, where are the data structure pointers, and what are their names. This information is essential for making sure that the data on the file are compatible with the version of the program you are using. Also, as you will learn later (see "Managing Changes" below) DOL provides an automatic version control, which allows to read old data files even if your classes or relations change. We are very proud of this feature, and believe that no other persistent system or OODBS have it, at least not in this extent.

The first objects recorded on the file are the key entries, then follow other objects. Each object is written to disk in two records: The first record describes the class of the object, whether it is a single object or an array, and its memory location. The second record is a plain copy of the object, byte-by-byte, as it is in the memory - including the internal pointers.

Saving to disk is relatively fast, because there is no formatting or massaging of data. Objects are stored as blocks of bytes. When restoring the objects from the disk (the *open()* function), objects are allocated in new places in memory, while keeping a table of their old and new address. After all objects are read to memory, DOL walks through them, and resets the internal pointers to new values (remember, DOL knows where the data structure pointers are, because *zzprep* created them. Replacement of the pointers is again quite fast, using hashing and only occassionally binary search.

Method 2: Saving in ASCII format

The use of *save()* and *open()* is the same as for the binary format. Also the internal algorithm is the same, but the storage to disk is a human-readable format, using ASCII. The main advantage of the ASCII format is that it is portable among different platforms. Note a potential problem with the interpretation of class members:

```
class X {
    ZZ_EXT_X
    char c;
    int i;
    char t[20];
    float f;
    ...
};
```

Is *c* a printable character or a short integer, and if latter, is it signed of unsigned? Is *t* a NULL ending string, or always 20 characters? What if *t* represents a name which includes blanks, such as "John Scrivens", how could we record that? Should f be printed in the decimal or exponencial format? Because only the program designer knows the meaning of the data members, it is much safer to let the programmer to specify this. When using the ASCII mode, the programmer must, at least in one *.h file, declare

```
    #define ZZascii
```

Also, the programmer must, for every class, provide a *ZZ_FORMAT()* statement. This statement describes, in the style used for *printf()*, the format for writing out all the object members. The format does not include the hidden (internal) data structure pointers. DOL knows about them, and knows how to interpret them in the serialization functions. Note that it is not necessary to store all members - temporary variables may be missing in the *ZZ_FORMAT()* statement and also in the disk image, resulting in a saving of the disk space.

DOL writes out each object as three ASCII records: The first record is a header similar to the header used in the binary saving, except that the values are in ASCII (readable numbers). The second record contains all the internal pointers, printed as integers. The third record contains all the members, printed in the format specified in the *ZZ_FORMAT()* statement.

The small inconvenience of writing the *ZZ_FORMAT()* statements gives is compensated by a great flexibility of this method. Since *zzprep* generates both the input/output serialization functions from the same *ZZ_FORMAT()*, they are guaranteed to match. It is much safer than to code serialization functions by hand. example:

```
#define ZZascii
#define ZZmain
  ...
class X {
    ZZ_EXT_X
    unsigned char c;
    int i,flg;
    char t[20];
    char *temp; // non-persistent pointer
    float f;
    ...
};
ZZ_FORMAT(X,"%d %o %s %f,c,i,t,f");
```

In the previous example, *temp* and *flg* are not stored on disk. Note the syntax slightly different from *printf*: Both the format and the parameter names are passed as a single text string. Also note that the order of the members does not have to be the same as in class X. For example, the following format provides the same functionality:

```
ZZ_FORMAT(X,"%s %d %o %f,t,c,i,f");
```

Exaple of using ZZ_FORMAT():

```
class Obj1 {
    ZZ_EXT_Obj1
    int a,b;
    float x;
    char c;
};
ZZ_FORMAT(Obj1,"%d %d %f %a,a,b,x,c");
class Obj2 {
    long int a;
    ZZ_EXT_Obj1
public:
    char *temp;
    float x;
};
ZZ_FORMAT(Obj2,"%e %lu,x,a");
class Obj3 {
    ZZ_EXT_Obj3
};
ZZ_FORMAT(Obj3,"");
ZZ_HYPER_SINGLE_RING(ring1,Obj1);
ZZ_HYPER_SINGLE_TRIANGLE(ring1,Obj1,Obj2);
ZZ_HYPER_SINGLE_LINK(ring1,Obj1,Obj3);
```

Note: When the object is used only as a part of the data structure, and has not attributes to be saved on the disk, an empty *ZZ_FORMAT()* still has to be provided. For example: *ZZ_FORMAT(myClass,"");*

Method 3: Memory blasting

This method of saving to disk is dramatically (order of magnitude) faster than the binary/ASCII formats. However, it requires a more careful use, and setting of some parameters. When designing new software, we recommend to use the binary or ASCII method of save first, and only when the program is debugged, switch to memory blasting by changing a few parameters.

In memory blasting, objects are allocated from pages of memory. A small portion of each page (about 3%) is used as for a binary map, which marks internal locations of the data structure pointers inside of all objects on the given page. These bits are set already when individual objects are allocated.

When saving data to disk, entire pages are directly copied (blasted) to disk, without looking at individual pages. This is extremely fast. Automatically allocated objects do not transfer to the disk, only objects allocated from the heap, using the operator *new*.

When retrieving data from the disk, entire page are copied back to memory. After that, DOL walks through the bit maps, and identifies data structure pointers without paying attention to in what objects they are. When the size of the pages is a power of 2, the conversion of the pointers requires only several computer instructions - no search or hashing is used. Detailed of this smart algorithm are described in Chapter 8 of "Taming C++: Pattern Classes and Persistence for Large Projects" by Jiri Soukup, published by Addison-Wesley 1994, ISBN 0-201-52826-6.

In order to invoke memory blasting, you just add two lines before the first call to *new*, usually in the begining of *main*:

```
util.mode(0,0,0,0);
util.blkAlloc(sizeEstimate,pageSzBits);
```

For more details about the *mode()* function, see below. *sizeEstimate* gives the maximum size of memory your data structures may use. As explained, the size of pages must be a power of 2. For this reason, *pageSzBits* provides the power. For example, if you want pages 1024 bytes each, and your overall data storage will not exceed 1,000,000 bytes, you invoke:

```
util.mode(0,0,0,0);
util.blkAlloc(1000000,10);  // 2**10 = 1024
```

WARNING: If you cannot estimate the memory needed for your data, you can specify sizeEstimate=0. In this case, DOL will assume that you plan to use all the address space (2**32). For example, if your page size is 2**10, this means that you can have up to 2**22 pages. Memory blasting reserves an array with one pointer (4 bytes) for each page, so unless your pages are large, this internal array can be quite large and block, unnecessarily, too much of your resources (e.g. an array with 2**22 entries, 4 bytes each, takes about 17MB).

NOTE: With memory blasting, key entries can even point inside of objects (to a base-object), which is not permitted in other saving methods.

For examples of memory blasting, see *test23f.c*, *test23g.c*, *test34b.c* and *test34c.c*.

Changing the mode of saving

Test programs *test34a.c*, *test34b.c*, *test34c.c*, and *test34d.c* demonstrate how - within one program run - you can open data in one format, and save it in another. All you have to do is to reset it by calling *util.mode(...)* with appropriate parameters before the next *save()* or *open()*. If one of the storage modes is ASCII, then *#define ZZascii* and *ZZ_FORMAT()* statements must be present.

```
// the following two lines invoke memory blasting
util.mode(0,0,0,0);
util.blkAlloc(sizeEstimate, pageSzBits);
...
util.mode(1,0,0,0); // invokes ASCII mode
...
util.mode(0,1,0,0); // invokes binary mode
...
util.blkFree(1);   // destroys (cleans up) pages used by memory blasting
util.clear(n,v,t); // cleans up the memory space used by the other two modes
```

Functions *mode()*, *blkAlloc()*, and *blkFree()* are useful in advanced storage of data, and their full syntax and meaning of individual parameters will be described later.

Saving individual objects

Except for the "List of functions and their syntax" at the end of this chapter, everything which will follow now can be considered the advanced use of the disk storage utilities. If you are a new user, read this part as an overview of what DOL can do, but unless you are experienced in using *save()* and *open()* with all the three basic saving (binary, ASCII, memory blasting), experimenting with the advanced features will bring you only frustration and a false impression that the library does not work.

There are situations, where it may be beneficial to save explicitely individual objects rather than relying on DOL to traverse all your data. For example, if all your data form a linked list, traversing this list and storing the objects one-by-one is faster than the sophisticated DOL algorithm which must cover some general, tricky cases. You also need less memory to perform this. The internal algorithm uses an array of pointers, one for each object. For large data sets, this can be a very large array.

Another situation where this may be useful is for saving updates of a small number of objects. We can take the advantage of the fact that both the binary and ASCII formats accept multiple copies of the same object on the file. If this happens, the latest copy is retrieved when you 'open' the file. A call to *save()* records each object on the file exactly once, except perhaps for the key entries, but consider the following situation. Assume we recently 'saved' the entire data image and, within the same run, a few objects change. If we record the new versions of those objects at the end of the file, we can avoid another call to *save()* - a significant performance enhancement.

If you alread called *save()*, the class tables and the key entries are already recorded at the beginning of the file. The only problem is that, in the default mode, the *save()* commend closes the output file. Since we want to add object at the end of the file, we must arrange that *save()* does not close the file, and then when we are finished with adding the few objects that changed, we must close the file explicitely.

```
util.mode(0,0,0,2); // last parameter specifies not to close the file
util.save("myFile",...);     // calling as usual
... update some objects, and record the update ...
util.close("myFile");
```

If you you are storing all the object individually, you have start with the key objects. When you record the first key object, DOL automatically initiates the file with the description of all your classes. In order to register key entries, call *ZZ_KEY_SAVE()*. For all other objects, call *ZZ_STORE()* or *util.deep()*. *ZZ_STORE()* stores a single object (shallow copy), while *util.deep()* stores all other objects that can be reached from it - through class membership, inheritance, or data structure pointers (more than a deep copy). At the end you have to close the file by calling *util.close(fileName)*.

ZZ_KEY_SAVE() and *ZZ_STORE()* are macros, and should be encapsulated under the classes that use them. For example:

```
class myObj {
    ZZ_EXT_myObj
public:
    void saveObj(char *fileName){ZZ_STORE(myObj,fileName);}
    void saveKey(char *fileName){ZZ_KEY_SAVE(myObj,fileName,this);}
};

myObj *p,*r;
r->saveKey("myFile");
p->saveObj("myFile");
```

The order in which the objects are saved is irrelevant, except that the key entries must be written out first. If you have only one key entry, or if all key entries are simple objects (no inheritance, no member objects), no calls to *ZZ_KEY_SAVE()* are required. The first object or objects automatically become key entries.

All objects must be written out; if you forget even a single object, *util.open()* will detect the error when restoring the data from disk. A missing object means that some pointers will lead to unrecognized memory locations.

As explained above, storing any object several times does not cause an error, even if the copies differ. The program eliminates duplications automatically (the last version written out is accepted as valid). However, multiple copies increase storage and increase the required CPU time. The option of being able to write out multiple copies is convenient, but should not be overused.

When saving a text string, use

```
char *fileName, *ptr; int n;
ZZ_OBJECT_SAVE(char,fileName,ptr,0); /* for NULL ending string */
ZZ_OBJECT_SAVE(char,fileName,ptr,n); /* for buffer of n bytes */
```

If your data has a single root, a call to *util.deep()* has essentially the same effect as *util.save()*, except that you must record the key entries first, then call *deep()*, and finally close the file.

Important: Retrieving data from disk is always done in a single command, *util.open()*, regardless of whether you stored individual objects with *ZZ_STORE()* or with *util.save()*.

Multiple files, private labels

In rare situations, the programmer may need more control in how the data is saved, for example splitting the data into several files, each with its own identification label or other information stored in the beginning of the file. The main steps involved in doing this are:

1. Call *util.mode()* with the last parameter *cntrl=2*, before the first command which writes to disk, and open the file.
2. Write your label or private information to the disk. Make sure you know where you are positioned on the disk.
3. Save the data using either *save()* or by *ZZ_STORE()*.
4. Close the file by *util.close()*.
5. Repeat these steps for other files.
6. When reading the data from the disk, use again *util.mode()* with the last parameter *cntrl=2*, open each file yourself, read the label or your private information, and then call *open()*.
7. After reading all the data back from the disk, pointers between the data stored in different files can be restored by calling *util.bind()*. This pointer conversion must be carefully planned, because *util.bind()* has the following limitation: If you are restoring data from the files *file1, file2*, and *file3* (in this order), pointers between *file1* and *file2*, and between *file2* and *file3* can be recovered, but not pointers between *file1* and *file3*.

Example:

The following code demonstrates how to add your own label at the beginning of the file, or/and some private data at its end. Note that *util.save(), ZZ_OBJECT_SAVE(), ZZ_STORE(), ZZ_KEY_SAVE(),* and *util.open()* must not be given file name, but rather a file pointer instead. The code below assumes the ASCII format. For the binary format, there would be *mode(0,1,0,1)*, and *fwrite(..,fp)* would be used instead of *fprintf(fp,...)*:

Saving data to disk:

```
util.mode(1,0,0,1);
FILE* fp=fopen("myfile","w");
fprintf(fp, ...); // adding label at the beginning of the file
util.save((char*)fp,...);
fprintf(fp, ...); // private data at the end of the file
...
fprintf(fp, ...);
fclose(fp);
```

Retrieving data from disk:

```
util.mode(...,1);
FILE* fp=fopen("myfile","w");
fgets(buff,BSIZE,fp); // read your label
sscanf(buff,...); // decode the line
util.open((char*)fp,...);
fgets(buff,BSIZE,fp); // read private data
sscanf(buff,...); // decode it
... // must be synchronized with how the data was created
fgets(buff,BSIZE,fp);
sscanf(buff,...);
fclose(fp);
```

Another example: Assume your data consists of several parts connected only by GENERAL_LINK's, and you want to save it in the binary format. You first call *util.mode(0.1.0.2)*, then call *util.save()* several times (one call for each part). Close the output file with *util.close()*:

```
util.mode(0,1,0,2);
util.save(...);
util.save(...);
util.close(...);
```

Active and passive blocks

The mechanism of storing data to disk is closely related to how the data is allocated. This applies in particular to memory blasting, where objects are allocated from pages of memory, which then are directly copied to the disk. The bit map which marks the locations of all pointers on these pages is set right when each object is allocated.

On the other hand, if different data sets are allocated from different pages, when we do not need one particular set any more, we can destroy its pages without looking at the individual objects inside them - assuming no other objects point into these pages. The destruction of objects in C++ can be quite expensive, and this simple technique can significantly improve the program performance. For example, when we applied this idea to the core software of a telephone switch, the overall speed of the switch improved three times!

Consider how we invoke the memory blasting mode:

```
util.mode(0,0,0,0);
util.blkAlloc(sizeEstimate,pageSzBits);
```

The call to *util.blkAlloc()* creates an 'active' block of pages. Word 'active' means that any call to *new()* will allocate the object from these pages. However, if we then call

```
util.blkActive("blk1",0);
```

It makes the block 'passive', and gives it the refence name, *blk1*. In order to allocate more objects, you need an active block. You either have to establish a new block of pages by calling *util.blkAlloc()* again, possibly with different parameters, or you can activate one of the passive blocks

```
util.blkActive("blk3",1); // blk3 must be a passive block we created earlier
```

Another way to activate a block is to call *blkActive(ptr,2)* where *ptr* is a pointer into one of its pages:

```
char *ptr;
...
util.blkActive(ptr,2);
```

In order to free the currently active block, call *util.blkFree(0).* Other uses of this command: *blkFree(1)* switches from the reserved pages to the normal C++ allocation, and should be used for temporary objects you don't need to store to the disk. When you want to switch from the normal allocation back to the currently active block, use *mblkFree(2).*

Any call to *util.blkActive()* brings you back to memory blasting mode, even if the previous call to *util.blkFree()* reset allocation to 'normal'.

When allocating objects from the active block pages, use either *new()* or *ZZ_OBJECT_NEW()* which also provides, on top of memory paging, a list of free objects by type.

Use *new()* and constructors as usual, but **never** place *ZZ_OBJECT_NEW()* into a constructor. Look at *test23b.c* for the correct way to use *ZZ_OBJECT_NEW().*

Sometimes it is useful to remember an important object (or the key entry) for each memory block (block of pages). An example is a situation, where you use several blocks, one for each design view. Function *blkUtil()* provides the mechanism to do this: it allows you to store, for each block, one utility hook (*void *hook*).

Coding your own serialization functions

When you declare *#define ZZascii* and, at the same time, a class has the *ZZ_FORMAT()* statement, *zzprep* automatically generates serialization functions for that class, and deposits the code into file *zzfunc.c*. In fact, it generates two input functions, and two output functions. Always, one function writes/reads the internal data structure pointers (you normally do not have no access to them, and may not even be aware they are in your objects), while the other function writes/reads the attributes which arem under your control. It is the latter function which is generated from the *ZZ_FORMAT()* statement.

If you declare *#define ZZascii*, but a class does not have any *ZZ_FORMAT()* statement, *zzprep* generates only the functions which write/read the internal pointers, but not the functions which handle your attributes. Since DOL needs these functions to perform saving and opening of the data, you have to code them yourself. These functions are called ***zz_inp_...()*** and ***zz_out_...()***, where the name of the class replaces the dots.

```
class Apple {
    ZZ_EXT_Apple
```

```
public:
    char colour;
    int numSeeds;
};
struct Plum {
    ZZ_EXT_Plum
public:
    float weight;
};
#define BSIZE 80
char buff[BSIZE];
```

Functions you have to code in this case are:

```
int Apple::zz_out_Apple(FILE *fp,Apple *p){
    fprintf(fp,"%c %d\n",p->colour,p->numSeeds);
    return(0);
}
int Plum::zz_out_Plum(FILE *fp,Plum *p){
    fprintf(fp,"%f\n",p->>weight);
    return(0);
}
int Apple::zz_inp_Apple(FILE *fp,Apple *p){
    if(!fgets(buff,BSIZE,fp))return(1);
    sscanf(buff,"%c %d",&(p->colour), &(p->numSeeds));
    return(0);
}
int Plum::zz_inp_Plum(FILE *fp,Plum *p){
    if(!fgets(buff,BSIZE,fp))return(1);
    sscanf(buff,"%f",&(p->>weight));
    return(0);
}
```

Rules for writing these functions:

- start with *zz_out_*.. or *zz_inp_*... (small zz, not ZZ);
- return(1) when end of file;
- print/read only the significant fields (temporary variables do not have to be reported);
- provide such functions for all structures that have *ZZ_EXT_*...
- input and output format must exactly match

Saving a union

If your class includes a union member, then *ZZ_FORMAT()* does not have enough flexibility to handle the situation, and you have to code your own *zz_inp_...()* and *zz_out_...()* for that class:

```
class  A {
    ZZ_EXT_A
    union {     // persistent, to be stored on the disk
        int k;
        char c[4];
    }uni;
    int ss;
```

```
        float ff;
        ...
    };

int A::zz_out_A(FILE *fp,A *p){
    fprintf(fp,"%d %d %f\n",p->uni.k;p->ss,p->ff);
    return(0);
}
int A::zz_inp_A(FILE *fp,A *p){
    if(!fgets(buff,BSIZE,fp))return(1);
    sscanf(buff,"%d %d %f",&(p->uni.k), &(p->ss), &(p->ff));
    return(0);
}
```

The union to be saved must not include a pointer to a persistent object.

If the union handles only temporary (non-persistent) values, *ZZ_FORMAT()* can be used as normal:

```
    class  A {
        ZZ_EXT_A
        union {    // temporary, non-persistent values
            int k;
            char c[4];
        }uni;
        int ss;
        float ff;
        ...
    };
    ZZ_FORMAT(A,"%d %f,ss,ff");
```

Format of the ASCII file

We encourage users neither to modify the format under which the ASCII data is stored, nor attempt to read it directly. The disk saving operation is more complex than it may appear on the surface. We include the format description mostly for curious, advanced users. If you are just starting with DOL, skip to the next section.

,

Before you continue reading, run one of the tests that save in ASCII format, for example *test23c.c* (includes inheritance) or *test0m.c* (no inheritance), and have a printout of the ASCII file ready when reading this explanation.

The ASCII file starts with the type table similar to the *ZZstrList[]* array which appears in your *zzincl.h* file. The difference is that in the ASCII file, all values are correctly set, while in *zzincl.h*, some of the values are just defaults that are overwritten by correct values at the beginning of the run. If you are curious about the meaning of individual fields in this table, look at the structure *ZZstrLST* in file *lib/bind.h*.

The record of the type table on the disk ends with the line starting *ZZendMark* ...

The second section describes the transparent pointers used by DOL organizations. These are pointers hidden under *ZZ_EXT_*.. statements, and you can find their names in the *zzincl.h* file which *zzprep* creates in your working directory. In the ASCII file, the name of each pointer is preceded by a special character, indicating its type (usually *a* for "automatic pointer"), and is appended by the character *[* and a number indicating the size of the array (1 for single objects). The size of this section is determined by the third last field on the line with *ZZendMark* above.

The third section describes the inheritance lattice of the types, and ends with a *ZZendMark* record. The content of each line is described in the structure *ZZtypeHier*, as shown in *lib/bind.h*. Note that each type in the type table has an index into this table (*inhInd*). If neither inheritance nor object members are used, and (*#define ZZ_INHERIT* is not present in *environ.h*), this section is missing (it is not used).

The fourth section contains, object by object, 2 or 3 records per object, depending on its type and content. The same object can appear several times, the last copy is considered valid when opening the file.

The first object record is a header which always contains 4 values: starting address, type index, size in bytes, array size (usually 1).

The second object record contains the transparent pointers, in the order shown under the *ZZ_EXT_* statement. This record is produced by the *function zz_opt_..* from your *zzfunc.c* file, and may be missing if there are no pointers.

The third object record contains integers, floats, and other members that you specified under the *ZZ_FORMAT* statement, or in your *zz_out_...()* function. If you coded the function, it can be in any file. If *zzprep* generated it from *ZZ_FORMAT()*, it was deposited into file *zzfunc.c*. This record again may be missing, if there are no other values to be stored than the transparent pointers.

In the case of an array, there is one header, and then pairs of lines corresponding to each object in the array.

The file ends with the header record (0 -1 -1 0).

Storing foreign objects

Another problem occurring in practice is that, in addition to regular classes and objects, the programmer wants to store objects from standard/private libraries. Classes for these objects cannot be modified by inserting pointers, or putting the *ZZ_EXT_..* statement into them. If these objects are not involved in any organizations except, perhaps, being a target for a *SINGLE_LINK* or *GENERAL_LINK*, then this can be done easily using the following method:

```
// ——— in the other library ———
struct foreign {
    int i,k;
};
// —— in your code ——
ZZ_EXT_foreign
typedef struct foreign foreign;
ZZ_FORMAT(foreign,"%d %d,i,k");
```

How does this work? The presence of *ZZ_EXT..* forces the code generator to register *foreign* as a recognized class. The class is not involved in any data organization, therefore it has no internal pointers or variables, *ZZ_EXT_foreign i*s empty, and can be anywhere, even outside the class. *ZZ_FORMAT()* automatically triggers the generation of appropriate IO functions, and the program can save this class like any other class:

```
foreign *f;
...
ZZ_OBJECT_SAVE(foreign,"myFile",f,1); // saves the object
```

User-controlled diskIO

When saving binary data to disk, the library is normally using fast binary IO - functions *open(), read(), write(),* and *close().* In exceptional situations, you may want to replace these functions by some other functions, for example when storing backup data to memory instead of to disk.

In order to do this, you have to do two things:

1. Code your own functions for diskIO, and pass them to the library through a call to *ZZinstallUserIO()*. Your functions must have types identical to the functions they are replacing. You may replace only *open&close* or only *read&write*, and use NULL for the remaining two functions (default will be used), but always the two functions which form a pair must be provided.
2. Transfer the control to your functions by calling *util.mode(0,2,0,0)* - this works for binary format only. If you want to return to the default library IO, call *util.mode(0,diskIO,0,0)* , with diskIO=0 or 1.

Example:

```
.....
FILE *myOpen(char *fileName,char *mode){ ... }
int myRead(FILE *fp,char *buff,int n){ ... }
int myWrite(FILE *fp,char *buff,int n){ ... }
int myClose(FILE *fp){ ... }
.....
ZZinstallUserIO(myOpen,myRead,myWrite,myClose);
util.mode(0,2,0,0);
```

Managing changes

As your program evolves, you may encounter two types of changes which may affect your ability to read your old data files:

1. **New version of DOL.** As DOL grows and improves, we try our best not to change the format of the disk storage. Over the 12 years of the DOL existence, it happened only twice but, theoretically, the possiblity is there.
2. **Different data structures**. New data organizations are added or some of the old data organizations are removed. The types or names of some data organizations change.
3. **Changes of classes,** adding or removing attributes, possibly adding new classes or discontinue old ones. Inheritance changes.

A change of DOL version is easy to handle. Function *mode()* allows you to set input and output format to two selected version of DOL.

When a new data organization is added, DOL handles the conversion. After retreiving the old data, the new organization is set as disconnected (or better to say: not connected yet).

When a data organization is removed, DOL again handles the conversion. Since the internal pointers which originally formed the organization do not exit any more, the organization naturally disappears.

Important: If any of the type parameters in a *ZZ_HYPER_..(id,type1,...)* statement change, *id* also must change. Essentially, the situation is handled as one data organization being cancelled, and another one introduced. The data organization disconnects, and is not properly copied into the new environment.

The number of key entries may be reduced. For example, if the data was saved with 3 key entries that correspond to types (Instance, Net, Instance), when reading the new data from disk only two key entries are returned (Instance, Instance). For more details see C examples in *test22a.c* and *test22b.c.*

Note that when the new organization does not include the object type originally specified as the key entry, *open()* returns *NULL* for the corresonding key entry

.

Netlist example - changing members and the organizations

(see also C-code examples in *test7c.c* and *test7i.c*):

This example describes connectivity of an electric circuit such as a VLSI chip, or a printed circuit board. Assume that the old data had 3 classes with the following data structures:

```
class Instance{
    float current;
    ZZ_EXT_Instance
};
class Net{
    ZZ_EXT_Net
};
class ActTerm{
    ZZ_EXT_ActTerm
    int x,y;
};
ZZ_HYPER_SINGLE_RING(iRing,Instance);
ZZ_HYPER_SINGLE_RING(nRing,Net);
ZZ_HYPER_SINGLE_TRIANGLE(byInst,Instance,ActTerm);
ZZ_HYPER_SINGLE_TRIANGLE(byNet,Net,ActTerm);
ZZ_HYPER_NAME(iName,Instance);
ZZ_HYPER_NAME(nName,Net);
ZZ_HYPER_TIME_STAMP(Net);
```

In the new arrangement, we decided to drop class ***Net*** with all organizations related to it. However, we want to add a doubly-linked ring for all ***ActTerms***, and also keep a name for ***ActTerm***. This is a major change of the architecture. Note that the attribute ***current*** has been removed from type ***Instance***, and a new attribute ***shape*** has been added to type ***ActTerm***.

```
class Instance{
    ZZ_EXT_Instance
};
class ActTerm{
    ZZ_EXT_ActTerm
    int x,y;
    int shape;
};
ZZ_HYPER_SINGLE_RING(iRing,Instance);
ZZ_HYPER_SINGLE_TRIANGLE(byInst,Instance,ActTerm);
ZZ_HYPER_NAME(iName,Instance);
ZZ_HYPER_NAME(tName,ActTerm);
ZZ_HYPER_DOUBLE_RING(aRing,ActTerm);
```

If you use the ASCII mode, DOL will be able to read the old data file into the new environment. It will generate the proper *RING* of Instances, the *TRIANGLE* of ActTerms by Instance, and the Instance name. Net related organizations will automatically be deleted. The RING *aRing* and the *NAME tName* will be initialized as unused. The value of *current* will be discarded, *x,y* will be re-generated correctly, and *shape* will be left uninitialized.

When you add members such as int, float, or char (for example *shape* in class *ActTerm*), the new members are not initialized, and are either 0 filled, or contain random values.

The order of members can be changed arbitrarily, but the first part of the *ZZ_FORMAT* statement (the actual format) must remain the same between the program that saved the data and the program which opens it later on.

When eliminating some data members, the action differs depending on whether you can anticipate the change already at the time

when you save the data:

1. If you know about the change at the time you create the data file (planned format conversion), simply drop the particular members from the *ZZ_FORMAT()* statement.
2. If you don't know about the change beforehand (handling older data without an access to the program which generated the file), give DOL a dummy variable into which it can read the members that are being discarded.

Example, not anticipating a change:

```
class myClass { // the original class
    int i,k,s;
    ZZ_EXT_myClass
    ...
};
ZZ_FORMAT(myClass,"%d %d %d,i,k,s");

// changed class, i is missing
class myClass {
    int s,k;
    ZZ_EXT_myClass
    ...
};
ZZ_FORMAT(myClass,"%d %d %d,k,k,s");
// reading the old i value into k, then overwriting it by the true k
```

Example, not anticipating a change:

```
class myClass { // the original class
    int i,k,s;
    float f;
    ZZ_EXT_myClass
    ...
};
ZZ_FORMAT(myClass,"%d %d %d %s,i,k,s,f");
// changed class, k and s are missing
union uni {
    int i;
    float f;
};
class myClass {
    int k;
    union uni u; // overhead to keep all garbage
    ZZ_EXT_myClass
    ...
};
ZZ_FORMAT(myClass,"%d %d %d %s,u.i,k,u.i,u.f");
```

For another example of this technique, see *test40a.c* and *test40b.c*. *test7e.c* shows another example of changing classes and their relations (schema migration).

Even though the main format used for managing changes is the ASCII format, some smaller changes are also automatically absorbed even for the **binary format**. The condition is that the order of attributes before and after the *ZZ_EXT_*.. statement must

remain the same. You may drop attributes only at the end of either section; new attributes must be added at their ends. For example, compare the following class definitions:

```
class ActTerm{  // original class
    ZZ_EXT_ActTerm
    int x,y;
    int termID;
};

class ActTerm{  // new class
    int shape;  // OK, new member at the end of the section before ZZ_EXT
    ZZ_EXT_ActTerm
    int y,x;    // problem, values of x,y will switch around
    // int termId; // OK, end of the section removed
};

class ActTerm{  // new class
    int shape;  // OK, added at the end of the section before ZZ_EXT
    ZZ_EXT_ActTerm
    int x,y;    // no problem
    int termID;
    int numOfPins; // OK, after the end of the section after ZZ_EXT_...
public:
    ActTerm(){shape=rotation=0;}
    int rotation; // OK, after the end of the section after ZZ_EXT_...
};
```

Note that when using the ASCII format, you control the read/write operation of attributes with the *ZZ_FORMAT* statement. The *ZZ_FORMAT()* statement or the custom coded *zz_inp_...()* and *zz_out_...()* refer to attributes by name; their order is irrelevant. When changing the organization, either code *zz_inp_.*. and *zz_out_*.. functions by hand, or use a different *ZZ_FORMAT()* for each version. DOL takes care of transparent pointers and other registered variables.
For example, in the situation above, when reading the old file and creating a new one, you need the following two functions. You are free to choose the order of attributes in *zz_out_ActTerm()*; the order in *zz_int_ActTerm()* must be the same as it was in *zz_out_ActTerm()* for the old data.

```
int ActTerm::zz_out_ActTerm(FILE *fp,ActTerm *p){
    fprintf(fp,"%d %d %d\n",p->x,p->y,p->shape);
    return(0);
}
int ActTerm::zz_inp_ActTerm(FILE *fp,ActTerm *p){
    if(!fgets(buff,BSIZE,fp))return(1);
    sscanf(buff,"%d %d",&(p->x),&(p->y));
    return(0);
}
```

Note that if the change involves a deletion of arrays or properties, dummy (unused objects) may be created when opening the old file. However, when saving the new organization, the dummy objects will disappear.

Version changes

The last section of this User's Guide, called **Revision history** lists new features and improvements since DOL Ver.5.0. It also indicates the compatibility of disk files with previous versions.

Only in two cases did the disk format change (Ver.1.65 and 2.0). No other changes in disk file format are planned in the foreseeable future.

DOL utility is most flexible. For example, if you can read data created under Ver.1.62 which is in the binary format, and write it to the disk in the ASCII format, using the current DOL version. All this is accomplished just by two *mode()* calls:

```
#define ZZascii
ZZ_HYPER_UTILITIES(util);
util.mode(0,1,162,0); // binary, buffered IO, ver.1.62
util.open(...);
...
util.mode(1,1,0,0); // ASCII, IO irrelevant, current version
util.save(...);
```

test21.c shows the situation when using the same data organization, but transferring between different versions of DOL.

Hierarchical types

This sections explains in more details the reasons for using *#define ZZ_INHERIT* in the *environ.h* file.

The algorithm for re-calculation of pointers during the *open()* operation is much simpler, if the pointers are guaranteed to point to the beginning of the allocated objects. Since persistent pointers in your objects can be implemented only through DOL organizations, you may ask when a pointer would not do that. There are two typical situations:

1. pointer to a base class;
2. pointer to a member inside an object;

Examples:

```
class D {
    ...
};

class C {
    ...
};
class B {
    ...
};
class A : public B {
    int ii;
    C cc;
    ...
};
ZZ_HYPER_SINGLE_LINK(cLink,D,C);
ZZ_HYPER_SINGLE_LINK(bLink,D,B);

D* d=new D;
A* a=new A;
B* b=a;
bLink.add(d,b); // setting pointer to the base class

C* c= &(a->cc);
```

```
cLink.add(d,c); // setting pointer to the member object
```

If we can prevent these two situations, we can remove *ZZ_INHERIT* from file *environ.h*, and even in the binary format *open()* will be noticeably faster. This means: lightweight objects (no inheritance), and no member objects.

For additional information, look at Chap.8.1.8. which also explains the use of *#define ZZ_INHERIT*.

Another example of the code where *#define ZZ_INHERIT* is definitely required:

```
class Root {
    ZZ_EXT_Root
};
class Shape {
    ZZ_EXT_Shape
public:
    virtual void anyFunction(); // see below
};
class Rectangle : public Shape {
    ZZ_EXT_Rectangle
    ...
};
ZZ_HYPER_SINGLE_COLLECT(col,Root,Shape);
ZZ_UTILITIES(util);
Root *rp; char *v[1],*t[1];
    ...
    v[0]=(char *)rp; t[0]="Root"; // set the key entry
    util.save("myFile",1,v,t); // saves all geometry
```

Important: If your class inherits from another class, it is critical that the base class has at least one virtual function. If it does not have one, the DOL internal type table is not constructed correctly. But why would you have a base class, if there are no virtual functions? Here DOL tries to be space efficient. The code generator (*zzprep*) analyses class definitions, and never expands them by introducing hidden virtual pointers, unless virtual functions are already used.

The same arrangement (requirement of at least one virtual function, perhaps a dummy one) applies to the automatic type detection - see Chap.11.15.

For examples of saving hierarchical types, see *test33a.c*.

Warning: If you save to disk, a class may inherit from a class, but not from a struct. For example

class A { ... };
class B: public A { ... }; // is fine

but

struct A { ... };
class B: public A { ... }; // will not work

Member objects

When storing objects of class B which includes members that are instances of another class A, one of the two conditions must be satisfied in order to provide automatic persistency. Either class A is a DOL registered class:

```
class A {
    ZZ_EXT_A // A is a registered class
    ...
public:
    A(){....) // anything in default constructor
};
class B {
    ZZ_EXT_B // B is a registered class
    A a; // instance of A is a member
    ...
};
```

Or the default constructor for class A must do nothing:

```
class A { // class without ZZ_EXT_..
    ...
public:
    A(){} // or it must not be defined
};
class B {
    ZZ_EXT_B // B is a registered class
    A a; // instance of A is a member
    ...
};
```

Making existing programs persistent

We often meet programmers who coded large projects without DOL - using their own data structures or other class libraries which either are not persistent, or are slow or have persistence which is difficult to use. Converison to DOL persistency can be done without converting all data structures:

1. Insert the *ZZ_EXT_..* statement into all your classes.
2. Replace all member pointers in your classes by either the *NAME* organization (for pointers to character strings), or by *SINGLE_LINK* (all other pointers).
3. Control the persistency usind DOL's *save()* and *open()*.

DOL also has macro *ZZ_PTR(id)* which used to simplify the conversion of older programs to DOL, especially when using the C language. This macro was used to replace all occurences of pointers in the original code. The idea is simple, but has not been used for several years. All recent conversions were done from scratch, because the programmers wanted to benefit not only from DOL persistency, but also from its fast data structures.

Example of converting a legacy code. Here is the original program:

```
class Dept {
    Dept *next;
    char *name;
public:
    char *nextName(void){return(next->list);}
    ...
};
```

Which can be converted to DOL, and then saved with *util.save()* or *ZZ_STORE()*. Here is the version saving individual objects:

```
class Dept {
    ZZ_EXT_Dept
public:
    char *nextName(void){return(ZZ_PTR(next)->ZZ_PTR(name));}
    void save(char *fileName){ZZ_STORE(Dept,fileName);}
    ...
};
ZZ_HYPER_SINGLE_LINK(next,Dept,Dept);
ZZ_HYPER_NAME(name,Dept);

int main(void){
    Dept* d=new Dept;
    ...
    d->save("myFile");
```

List of functions and their syntax:

Note: Commands with names entirely in capital letters are formally macros, usually hiding one or a few function calls. Remaining commands are methods of the UTILITIES class. The definition of this class hides under ZZ_HYPER_UTILITIES().

ZZ_HYPER_UTILITIES(id);	declaration of the UTILITIES class, creates also its instance, *id*
*void id.close(char *fileName)*	writes out an end-record, and closes the output file (several files may be open simultaneously).
*void id.save(char *fileName,int num,char *keyEntries[],char *keyTypes[])*	starts from the given key entries [total of *num* key entries is given], and saves the entire data set to the file.
*void id.deep(char *fileName,void *obj,char *type)*	saves one object with all connected objects, and can be thought of as a simplified case of *save()*. However, this command is different in how it opens and closes the output file, and should be used in special situations only.
*ZZ_STORE(TYPE,char *fileName)*	is a macro which can be encapsulated in a function to save a single object.
*ZZ_OBJECT_SAVE(TYPE,char *fileName,char *obj,int num)*	is a more general macro for the same purpose, which allows you to write out arrays of objects or text strings. *num* is the number of objects (size of the array). If *TYPE=char* and *num=0, t*he object is treated as a NULL ending string. If *TYPE=char* and *num0***,** the object is treated as a block of *num* bytes.
*void id.open(char *fileName,int num,char *keyEntries[],char *keyTypes[])*	retrieves the entire data set from the given file, and restores all pointers. The key entries are returned in their original order [total of *num* key entries is expected].

*void id.clear(int num,char *keyEntries[],char *keyTypes[])*	starting from the given key entries [total of *num* key entries is given], this command collects all of the objects in the given organizations, and deallocates them without testing the pointers. This command is disabled (it does nothing) in the binary and ASCII modes when *ZZ_INHERIT* is defined in the environ.h file (most C++ programs). You have to traverse the organization and deallocate objects individually instead. This function always works in the memory blasting mode.
void id.mode(int ascii,int diskIO,int version,int fileCntrl)	resets the format for SAVE/OPEN *ascii*=0 for binary format, =1 for ASCII format, *diskIO*=0 for direct diskIO (write/read), =1 for buffered diskIO(fwrite/fread), =2 for user-controlled diskIO. *version*=version number as an integer, for example 164 for version 1.64, or 200 for version 2.0. You can also use version=0 for the current version. *fileCntrl*=0 automatically closes the output file, =1 for outside file opening/closing, =2 does not close the file.
char id.bind(char *oldPtr)*	converts the old pointer into the new location, after ***open()*** has been called. This command is tricky to use, and is not intended for novice users. It allows you to reset connections between data sets stored in different files.
void id.keepTbl(void)	has to be called prior to *open()*, if you plan to convert old-to-new pointers in a semi-automatic way. The result is that the internal conversion tables remain in memory after the program returns from *open()*. The tables stay in memory until you call *freeTbl()*.
void id.freeTbl(void)	frees the internal tables used for the conversion of pointers.
void id.blkAlloc(int size,int bits)	reserves either one big block of memory of *size* bytes (if bits=0), or opens a set of pages that correspond to *bits* (for example *bits=8* represents pages of 256 bytes). In the second case, size is the overall (very rough) limit on the memory to be used.
void id.blkFree(m)	for *m*=0 frees the currently active block of pages, for *m*=1 switches to regular (plain) allocation, for *m*=2 returns to block allocation.

*void id.blkActive(char *blkName, int aCode)*	For *aCode=0*, it makes the current block of pages passive, and stores it under the given name (in memory, not on disk). For *aCode=1*, it restores the given block as active. For *aCode=2*, it restores the block identified by the address (not by the name). In this case, blkName represents not a character string, but a pointer anywhere into the block to be activated.
char id.blkUtil(char *blkName,void **hook,int mode)*	searches for a block identified by its name or by the address anywhere inside the block (*blkName*). Then it sets or retrieves a utility hook to user data: *mode=0* sets hook, *blkName* represents block name, *mode=1* sets hook, blkName represents an address, *mode=2* gets hook, *blkName* represents block name, *mode=3* gets hook, *blkName* represents an address. In all cases, the function returns the name of the identified block, NULL if the block has not been found.
ZZ_FORMAT(myObj,aFormat)	declares the ASCII format for those attributes of *objType*, which should be stored/restored from disk automatically. This command is an instruction for the code generator, which generates the functions *zz_inp_objType()* and *zz_out_objType()* in the file *zzfunc.c*. Note that *aFormat* is the same as the regular format used for *printf()* and *scanf()*, except that the attribute list is also within double quotes. The attributes are listed directly without any pointer (or object) reference, and can be listed in any order. Temporary variables that do not have to be stored may also be omitted; in some situations, this can save a considerable amount of disk space. If there are no attributes to be saved, use an empty string, for example, ZZ_FORMAT(myObj,""). If you save in ASCII format, there must be a *ZZ_FORMAT()* statement for every class with *ZZ_EXT_*... Classes without the *ZZ_FORMAT()* statement do not get stored!

Comments:

1. If you use the binary and ASCII format for saving to disk, we recommend buffered IO (*util.mode()* option *diskIO*=1). It minimizes disk access. Instead of writing individual objects to disk, the objects are written into an internal buffer, which automatically moves to disk when full or when the file is closed.
2. *aFormat* in *ZZ_FORMAT()* must be an explicitly typed text string, not a run-time generated string.
3. If you use single byte variables to store small integers instead of real characters, use a special format (%a) which writes/reads them as numbers. This format is DOL specific, it is not supported by C/C++ compilers.

If you want to use %d, but cast the member as some unusual type, use %a and cast for the member, as in the example below.

```
ZZ_FORMAT(myObj,"%a,(short)w");
```

Previous Section 13.2 Memory Management

Next Section 13.4 Collecting Objects

14. DEBUGGING WITH ORGANIZED C

14.1 Using a Regular Debugger

All OrgC++ calls, whether they are macros or compiled functions, are encapsulated in hyper-classes. You can use your regular debugger to debug the program. You should not debug OrgC++ functions just as you do not debug a C++ runtime library. If you step through the program, skip over calls with the prefix ZZ.

If you insist on looking inside ZZ macros, read Chap.16; it explains the whole setup.

Automatically generated pointers that hide under *ZZ_EXT_* statements cannot be accessed from the outside - they are in the private part of your classes. Even though you can find the names of these pointers in *zzincl.h*, do not try to access them directly. For example, if you want, as part of your debugging, to find the next object on a ring, use the function *fwd()* and not a direct pointer jump. Always use only OrgC++ calls to access the data, even when you are debugging.

For an experienced C or C++ programmer who is used to working with pointers all the time this may be shocking news. Not use pointers, even when debugging? Simply consider the pointers that form the data organization to be off-limits. The library has been debugged and well tested. As you will see with time, if you get any errors, they will be most likely in your code or in how you call OrgC++ functions, not inside them. If you find anything wrong or strange, call Code Farms.

It is a good strategy to design your program with testing in mind. As you code the program, prepare utility functions that print subsets of your data, and insert calls to these functions as needed during debugging. With all the iterators that OrgC++ provides, coding such functions is trivial.

This strategy is usually more effective than stepping through the code and checking individual pointers. Remember that, when using OrgC++, you work with data of a much higher level. You should not be concerned about individual pointers.

All this advice assumes, of course, that you are debugging a program which uses the existing OrgC++ library. If you are testing a new organization or functions that you developed yourself and added to the library, the situation is different. Chap.16.5 describes how to proceed in such a situation.

14.2 Interactive Browser

Organized C allows you to display internal data and traverse it object by object. This can be done either in standalone mode, for example when examining disk files, or from inside your program as part of

debugging.

As we explained above, there should never be a situation which would require you to debug individual relations between the data. However, for the debugging of your application, it may sometimes be useful to browse through the data.

Using this feature is very simple. You need

#define ZZascii

and a *ZZ_FORMAT()* statement for each class that has *ZZ_EXT*.

If your program saves in binary format, use *util.mode(0,..)* before any calls to *save()* or *open()*.

At the place where you want to invoke interactive debugging, call *void util.debug(void *ptr,char *type)*. For example, the sample printout below comes from a program which deals with classes *Block*, *Net* and *Pin*.

```
ZZ_HYPER_UTILITIES(util)
Block *bp;
...
util.debug(bp,"Block");
```

This call puts you into an interactive mode where, starting from a given object, you can traverse the adjacent objects by typing the code number of the pointer. Code -1 returns you to the previous object (reverse move), and can be repeated any number of times. Code 0 exits the interactive part and transfers control back to your program.

As you can see from the example below, the program displays both the pointers and the attributes of the objects.

The first letter of each pointer name indicates its purpose - it matches the character used in the tables from the '*zzmaster*' file. For example, c=child, p=parent, f=forward, b=backward, n=name, and so on. The second part of each name is the name of the organization, as declared in the *ZZ_HYPER_..* statement. The arrow indicates to what object type the pointer is leading.

The following example corresponds to the problem from *test0d.c*, where objects *Block*, *Net* and *Pin* are connected in a netlist:

```
object=151755162 type=Block:
1: 1517551652 cbyBlock->Pin
2: 1517552080 fbChain->Block
3: 1517551648 nbName name=B1
x1=10 y1=0 x2=15 y2=5
```

```
select pointer(-1 retrace, 0 exit):1

object=1517551652 type=Pin:
1: 1517551760 sbyNet->Pin
2: 1517551692 sbyBlock->Pin
3: 1517551676 pbyNet->Net
4: 1517551624 pbyBlock->Block
x1=5 y1=2 x2=5 y2=2
select pointer(-1 retrace, 0 exit):3

object=1517551676 type=Net:
1: 1517551652 cbyNet->Pin
2: 1517551940 fnChain->Net
3: 1517551688 nnName name=N1
select pointer(-1 retrace, 0 exit):-1

object=1517551652 type=Pin:
1: 1517551760 sbyNet->Pin
2: 1517551692 sbyBlock->Pin
3: 1517551676 pbyNet->Net
4: 1517551624 pbyBlock->Block
|x1=5 y1=2 x2=5 y2=2
select pointer(-1 retrace, 0 exit):4

object=1517551624 type=Block:
1: 1517551652 cbyBlock->Pin
2: 1517552080 fbChain->Block
3: 1517551648 nbName name=B1
x1=10 y1=0 x2=15 y2=5
select pointer(-1 retrace, 0 exit):0
```

Example:

The use of the browser is the same in C and C++. For an example in C, look at *test24.c*.

« Chapter 13: Memory Management and Saving on Disk

Chapter 15: Reporting Errors »

15. ERROR MESSAGES

When you program with OrgC++, errors are reported at three levels:

Level 1: The class generator (*zzprep*) reports some errors, referring to the line numbers of the code that you ran through it. In most cases, this is only your header file with the definitions of organizations (*ZZ_HYPER_ statements*). The rule of thumb is that all OrgC++ calls which start with the prefix *ZZ_* should run through the class generator. If you have several header files, concatenate them into a single file, and run the class generator on that file.

Level 2: The compiler reports additional errors. The class generator is designed to force the compiler into more checking. With OrgC++, it takes a bit longer to get the code through the compiler, but the resulting program is safer to run.

Level 3: At run-time, OrgC++ provides more checking, mainly to prevent dangling pointers.

Normally, messages at all levels come out on standard output (*stdout*) - see below. When you receive your copy of OrgC++, the output language is set to English. You can convert level 1 and 3 messages to any other language, if you translate all the messages in the file *orgC/lib/msgs.c* and then recompile the entire library, including the class generator (see Chap.2 on how to recompile the library).

In addition to run-time error messages, programs coded with OrgC++ may set a special error flag that may be checked by your program. The flag is binary coded, and can be accessed through the following function: *int util.error(void)*;

```
ZZ_HYPER_UTILITIES(util);
util.strAlloc(...);
if(util.error()){
    printf("error=%o\n",util.error());
    return;
}
```

The error condition is bit encoded (shown as octal here):

01=allocation error
02=index out of range
04=using an organization that has not been formed
010=object not disconnected, or not compatible
020=internal algorithm failure
040=wrong structure
0100=wrong file or unable to open a file

The error condition can be set to the no-error state by calling *void util.ok(void)*.

If you don't want your messages on the standard output (*stdout*), provide your own error handling function somewhere in your code. The default version of *zzReportError()* is in file *orgC/lib/heading.h* and consequently also in file *orgC/zzcomb.h*. If you include *#define ZZcallBack* in your environment file, the default version will be disabled and prevent multiple declaration of this function. The default version contains a single call to *printf()* which you can replace by a *fprintf()*, or turn it into a call-back function that handles the errors.

In particular, when using Microsoft Visual C++, the *printf(...)* statement inside *zzReportError()* can be replaced by *TRACE(...)*. Sending messages to *stdout* is usually not practical in the visual environment. Note however, that *TRACE(...)* displays the messages only in the debugging mode, not in the release mode. That is not sufficient since OrgC++ may display run-time error messages which are vital for detecting serious errors. If you don't want error codes which include both alpha and numeric characters, use function *errorCodeConvert()* which converts the error code string into a unique long integer (file *msgs.c*).

Installation Error 2104

For instructions see Chap. 2.6.

Pointer References in Error Messages

Some run-time error messages give you an indication which data structure (and which internal pointer) has been corrupted or is causing the problem. The best way to explain this is on an example.

Example 1:

A program using objects of class Node, issues the following message:

> cannot add to dsNodes, object 7950688 not disconnected

You look at the ZZ_HYPER_.. statements and search for *dsNodes*

```
ZZ_HYPER_SINGLE_AGGREGATE(dsInElem, NetwElem, Node);
ZZ_HYPER_DOUBLE_AGGREGATE(dsNodes, Layer, Node);
ZZ_HYPER_SINGLE_AGGREGATE(dsSegments, Layer, Segment);
....
```

Now it is clear, that the program tried to add a Node with address 7950688 to aggregate dsNodes, but the Node is already in this aggregate. This operation would likely destroy the integrity of your data, and was not performed. The address of the object is usually irrelevant. You have to check your code in the place where it adds objects to dsNodes. Search for all occurences of *dsNodes.add(...)*.

Example 2:

The same program issues the following message:

> errLevel=8 errCode=SD3: cannot free object=Node, pointer No.0 not disconnected

This message may occur, if you requested testing of objects before destructing them, and you placed ZZ_CHECK() into the destructor:

> ~Node(){ZZ_CHECK(Node);}

The error message tells you that when your program tried to destroy a Node, the Node was not disconnected from all data structures. This will cause a serious corruption of your data, and the program will likely crash later on. Unfortunately, once the destruction process started, the library cannot stop it, but at least you know what happened, and the tedious debugging which would normally follow is prevented.

The message tells you that the internal pointer index=0 on object Node is not disconnected. In order to interpret this, you look at file *zzincl.h*, search for *ZZ_EXT_Node*, and here is what you may see:

```
#define ZZ_EXT_NodeLink     \
    Layer *ZZpdsNodes;      \
    Node *ZZfdsNodes;       \
    Node *ZZbdsNodes;       \
    Node *ZZfdsNodeLink;    \
    NodeLink *ZZsdsSameCost;\
    ...
```

The names of the pointers combine the name of the data structure with a one letter code expressing the purpose of the pointer. For example p=parent, c=child, f=forward link, b=backward link, n=name. The first pointer relates to dsNodes, so you see that the Node is still not disconnected from this aggregate. The code 'p' indicates that it is the parent pointer (and possibly some other pointers) which is not NULL at the time of the destruction.

« Chapter 14: Debugging with OrgC++ Chapter 16: Adding New Features »

16. ADDING NEW FEATURES

If you are just starting with OrgC, skip this entire chapter. You may need this information later, but only if you decide to add functions to the library.

There is a big difference between adding new features to OrgC++, which is based on C++ classes, and to OrgC, which is based on a combination of macros and compiled functions. In OrgC++, all functions related to the same organization are in one file, which describes the hyper-class of the organization. In OrgC, each macro or compiled function has its own file. The equivalent of hyper-classes is implemented through tables stored in the file *zzmaster*.

If you look at any of the files in *orgC/macro/hyp**, you will see that numerous ZZ-prefixed macros are used inside the hyper-class declarations. This may create the false impression that the hyper-classes are not coded in pure C++, and that they are only C functions called through a C++ interface.

These macros do not have to be there. If you take the macros from whatever file they are, and insert them into the hyper-class definition, you will get "normal" C++ code. The reason for using the macros is to maintain compatibility between C and C++ versions of Organized C, and also to simplify maintenance of the whole system. When you add a new hyper-class, you don't have to use ZZ-macros, except for two or three macros that create the pointer and type names. (*ZZFP(), ZZFN()*).

16.1 ADDING A NEW ORGANIZATION

When you want to add a new organization to OrgC++, you have to do two things: You plan the organization and register it in *orgC/macro/zzmaster* (Chap.16.1); then you code the hyper-class and access functions (Chap.16.2).

Decide how many transparent pointers and other variables the new organization will need, then edit the file *orgC/macro/zzmaster*, find the end of the section that starts with *ZZorganization*, and append one line there. The line must contain the following information:

- order index;
- new organization name;
- p1,p2 = upper and lower index of pointers in the *ZZpointer* section;
- nInp = number of object types + 1 = num.parameters of *ZZ_HYPER_..();*

Add pointers/variables required for the organization at the end of the *ZZpointer* section. Each pointer/variable needs the following fields:

- order index, must agree with the cross reference from *ZZorganization*;
- *toUse* and *pointTo* are type definitions, referring to the order of parameters in *ZZ_HYPER_*..

For example, if you plan to add *ZZ_HYPER_NEW_ORGANIZATION* (*id,type1,type2*), then a pointer which is on object *type1* and points to object *type2*, will set *toUse=1*, *pointTo=2*, while a pointer on object *type2* and pointing to a character string (*char* *), will have set *toUse=2*, *pointTo=0*.

- pointer type, one character field which determines the type of variable: 'a'=automatically managed pointer, 's'=string (*char**), 'c'=character, 'i'=integer, 'f'=float.
- pointer name, must be ZZ plus one lower case letter. Try to use meaningful names, such as *ZZp* for parent, *ZZf* for forward, etc. You can use the same name in different organizations, OrgC++ is protected against conflicts between different organizations. For your own reference, you may add a comment at the end of each line, which explains the meaning of the pointer (examine the example below, or the actual file *zzmaster*).

If you look into the actual file *zzmaster*, you will see that some pointer/variable names do not fit the recommended 3 character pattern, for example *ZZprop*. These pointers belong to hardwired organizations, which are subject to special rules.

Example:

The OrgC++ library contains *SINGLE_TRIANGLE*. Assume that you want to work with a hierarchy, where each level forms a doubly-linked ring, and you want to introduce a new organization called *DOUBLE_TRIANGLE*.

The organization will be declared by

ZZ_HYPER_DOUBLE_TRIANGLE(id,topType,botType);

and your original *zzmaster* file looks like this:

```
/* ind organization     p1      p2      nInp */
ZZorganization {
0       SINGLE_RING     0       0       2
.......................
12      LIFO            20      20      2
13      SINGLE_TREE     21      23      2
}
/* ind usedOn pointTo arrSize Type ptrCode */
ZZpointer {
0       1       1       1       a       ZZf
...................
22      1       1       1       a       ZZc
23      1       1       1       a       ZZs
}
```

You will add the new organization (index 14), with 4 new pointers: *ZZp*=parent, *ZZc*=child, *ZZf*=forwardSibling, *ZZb*=backwardSibling.

For example, the *parent* pointer is on the *topType*(2) and points to the *topType*(1), *ZZp* has *usedOn=2 pointTo=1*.

The new *zzmaster* file:

```
/* ind organization     p1      p2      nInp */
ZZorganization {
0       SINGLE_RING     0       0       2
.......................
12      LIFO            20      20      2
13      SINGLE_TREE     21      23      2
14      DOUBLE_TRIANGLE 24      27      3
}
 /* ind usedOn pointTo arrSize Type ptrCode */
ZZpointer {
0       1       1       1       a       ZZf
.....................
22      1       1       1       a       ZZc
23      1       1       1       a       ZZs
24      1       2       1       a       ZZc     // child
25      2       1       1       a       ZZp     // parent
26      2       2       1       a       ZZf     // forward
27      2       2       1       a       ZZb     // backward
}
```

Recommendation:

In your new organization, plan on using a ring-type arrangement where valid pointers cannot have a NULL value. If you do so, your program will be better protected at run-time.

16.2 CODING A HYPER-CLASS

The whole hyper-class will be in one file. All files containing hyper-class C++ definitions start with *hyp*.... You may follow the same convention, or use a special prefix, for hyper-classes which you designed yourself. The file, as with all files in the macro directory, will contain both the documentation and the computer code, separated by special comment lines:

```
/* ======================================== */
   ... documentation ...
/* ======================================== */
   ... first part of the macro,
       which also will appear
```

```
        in the documentation ...
/* ++++++++++++++++++++++++++++++++++++++++ */
   ... remaining part of the macro ...
/* ________________________________________ */
   ... compiled functions ...
```

The first line must be /* ====... , and not a blank line, otherwise the program treats the file as encrypted, and produces complete gibberish.

Note that the separation mark /* __... uses the underscore and not dash.

If your hyper-class uses only in-line functions, the last separation line and the last section can be omitted.

Note that the matching between the library and hyper-class declarations in your program will be done by the class generator (preprocessor). For example, if you declare

ZZ_HYPER_DOUBLE_TRIANGLE(myTriangle,topType,botType);

the file *zzincl.h* will contain the following statements:

```
#define ZZ1myTriangle   topType
#define ZZ2myTriangle   botType
#define ZZ_EXT_topType \
....
botType *ZZcmyTriangle;\
....
#define ZZ_EXT_botType \
....
topType *ZZpmyTriangle;\
botType *ZZfmyTriangle;\
botType *ZZbmyTriangle;\
```

zzincl.h gives you access to macros that paste arguments:

form pointer: example:	*ZZFP(A,B)* *ZZFP(ZZf,myTriangle)*	creates AB creates ZZfmyTriangle
form type: example:	*ZZFT(A,B)* *ZZFT(1,myTriangle)*	creates ZZAB creates ZZ1myTriangle= topType
nameToString: example:	*ZZ_STRINGIT(A)* *printf("%s/n",ZZ_STRINGIT(myTriangle));*	creates "A"

You can find the pasting macros in *orgC/lib/heading.h.* Most OrgC macros (for use in C) are coded in two levels. The first macro (*ZZ_..*) sets up pointer and type names; the second macro (*ZZZ_..*) does the actual calculation and uses simple and meaningful names for variables and pointers. All this name conversion is not

needed in C++.

Example:

Continuing the example from the previous chapter, we will present segments of the hyper-class for *DOUBLE_TRIANGLE*. The hyper class will be coded in the form of a parametric macro, stored in the file *hypdtria*. OrgC++ uses file names with 8 characters or less to maintain portability between UNIX and PC DOS. For the same reason, there should be no special use of upper/lower case in the file name.

```
/* ====================================
   your documentation in nroff -me format,
   including:
      definition of parameters
      description of both the organization and functions
      possible errors or exceptions
      example
==================================== */
#define ZZ_HYPER_DOUBLE_TRIANGLE(id,pType,cType) \
/* ++++++++++++++++++++++++++++++++++ */ \
class ZZFP(ZZH,id) { \
    friend class pType;\
    friend class cType;\
public: \
    cType *fwd(cType *s){return(s->ZZFP(id,ZZf);};\
    cType *bwd(cType *s){return(s->ZZFP(id,ZZb);};\
    pType *par(cType *s){return(s->ZZFP(id,ZZp);};\
    cType *child(pType *p){cType *c;\
        c=p->ZZFP(id,ZZc); return(fwd(c))};\
    ....
    void del(cType *p);\
};\
ZZ_EXTERN ZZFP(ZZH,id) id; \
class ZZFN(id,iterator){ \
    cType *start;\
    cType *nxt;\
public:\
    ZZFN(id,iterator)(pType *p)\
        {nxt=NULL; if(p)start=id.child(p); else start=NULL;};\ cType*
operator++(){cType *p; if(nxt==start)p=NULL;\
    else {if(!nxt)nxt=start; p=nxt; nxt=id.fwd(nxt);} return(p);};\
cType* operator==(){cType *p; if(nxt==start)p=NULL;\
    else {if(!nxt)nxt=start; p=nxt; nxt=id.bwd(nxt);} return(p);};\ };
/* ____________________________________ */
void ZZH$ :: del(ZZ2$ *t){
    if(t && t->>ZZp$){ // if not connected, do nothing
        if(t->ZZf$==t)t->ZZp$->ZZc$=NULL; // only child
```

```
            else {
                if(t->ZZp$->ZZc$==t)t->ZZp$->ZZc$=t->ZZb$;
                t->ZZb$->ZZf$=t->ZZf$; // relink neighbours
                t->ZZf$->ZZb$=t->ZZb$;
            }
            t->ZZp$=NULL; // disconnect parent
            t->ZZf$=t->ZZb$=NULL; // disconnect neighbours
        }
    }
```

Note several important details of how to deal with pointer/type names:

In the first part (class and iterator declarations which are included through the *zzincl.h* file) two macros are used: *ZZFP(id,ptrName)* creates the actual pointer name from the *ptrName* registered in the *zzmaster* file by concatenating the two names. *ZZFN(id,name)* concatenates the two names with an underscore (_) in the middle, and that is used here to create the name of the iterator class.

In the second part (compiled functions which are deposited into the file *zzfunc.c)*, the $ character is used to parametrize the pointers and types. The class generator will replace *ptrName$* by the name of the actual pointer, *ZZ1$* by the type of the first parameter (*pType* here), and *ZZ2$* by the type of the second parameter (*cType* here).

The parametrization logic is simple, and the meaning of the pointers easy to follow. Classes coded in this style cause no special difficulties during debugging.

Note that if the C implementation of the *DOUBLE_TRIANGLE* already existed, we could take a shortcut and code the *del()* function like this:

```
ZZH$ :: del(ZZ2$ *t){ZZ_DELETE_DOUBLE_TRIANGLE($,NULL,t);}
```

which would have no impact on code complexity, run-time performance, or the purity of C++ implementation.

Record the ZZ_HYPER_ macro in the *zzmaster* file:

Anywhere in the section of *ZZfunction* (here the index is not important) enter the following line:

```
ZZhyp2Tri ZZ_HYPER_DOUBLE_TRIANGLE hypdtria 13 -1 h
```

where

ZZhyp2Tri is the short name for the macro;
ZZ_HYPER_DOUBLE_TRIANGLE is the long name;
hypdtria is the file where the macro is stored;
13 is the organization index;
-1 is the index into the list of support files, -1 mean none;
h is the type of macro/function, here h=hyper-class.

For hyper-classes you do not need any support files.

UTILITY MACROS:

If you need a general utility function which does not relate to any particular organization, add it to the *UTILITIES* hyper-class (file *hyputil*).

OrgC++ contains three special hardwired classes: SELF_ID, TIME_STAMP, and PROPERTY, which are neither regular base classes nor hyper-classes. Each may appear on any object not more than once and, with the exception of PROPERTY, the variables that form them need special treatment. All three organizations are associated with the object itself (just like a base class), and have no hyper-class id.

For example:

```
class Block {
    ZZ_EXT_Block;
...
};
ZZ_HYPER_TIME_STAMP(Block);
Block *bp;
...
bp->setTime();
```

You cannot add such special classes to the OrgC++ library. However, since only the object itself is involved, you can achieve the same effect by using plain inheritance outside of OrgC++.

Creating new *zzcomb.h:*

After you introduce a new organization or a new function, the file *orgC/zzcomb.h* must be updated. When you run *zzprep* hyper-class declarations are taken from this file and not from individual files in the *orgC/macro* directory. The file *zzcomb.h* combines all macros stripped of comments. The file *zzfunc.c* is created any time you run the class generator.

To update *zzcomb.h*, do this:

cd orgC
zzcomb

If you forget to update *zzcomb.h*, your first run of *zzprep* will take a long time. The program detects that *zzcomb.h* is out of date, and invokes the *zzcomb* run automatically.

16.3 ADDING A NEW FUNCTION

When you want to add/modify a function to an existing hyper-class, simply modify the file *orgC/macro/hyp...*, and re-run *zzcomb* in the *orgC* directory.

16.4 ADDING A FUNCTION TO THE COMPILED LIBRARY

In some situations, you may want to move some functions into a compiled library. For example, you may have to hide (for commercial reasons) the critical part of your code from the programmers who will use the class. Using the library will also avoid unnecessary re-compilation of already tested code, when debugging the program.

This is how to add a new function to the library:

In UNIX:

```
cc -c newFun.c
ar r zzlib.a newFun.o
ranlib zzlib.a
```

In DOS (TurboC++):

```
tcc -mm -c -Lc:\turboc\startups;c:\turboc\lib -Ic:\turboc\
    include newFun tlib zzlib /C -+ newFun
```

16.5 DEBUGGING NEW FUNCTIONS AND CLASSES

The most convenient way to debug a new hyper-class is to run *zzcomb* and the class generator once, and then continue working with *zzcomb.h* and *zzfunc.c*, making modifications directly in them until everything works as expected. The file *zzfunc.c* has no macros inside, and is easier to read. You can recognize the meaning of all the pointers by the beginning of their names. If you have problems with the parametric definition of the hyper-class in *zzcomb.h*, replace the pasting macros, temporarily, by actual names. This is very simple, for example, if you have id=*myRing*, then *ZZFP(id,ZZf)* becomes *ZZfmyRing*, and *ZZFN(id,iterator)* becomes *myRing_iterator*.

When you finish debugging, recode *orgC/macro/hyp*...back into parametric form, rerun *zzcomb*, and re-test the whole setup.

IMPORTANT:

There are several serious reasons why OrgC++ implements hyper-objects through parametric macros. Templates introduced with C++ Ver.2.1 are limited to the use of type only, and do not allow argument pasting which provides close interaction between application code and class generator.

The simple linkage between a hyper-object library and user's code cannot be implemented with existing C++ tools, and we believe that our approach of using macros is more practical than writing a new C++ compiler which would compile data structures directly as a part of the language.

Watch for some typical errors when coding macros:

You need "\" at the end of each line (except for the last line) in the declaration of the hyper-class and (possibly) in the iterator.

Remember the different style of parametrization in the macro part: *ZZFP(id,ZZf)*, compared with *ZZf$* in the function part.

You have to generate one global instance of the hyper-class, by using *ZZ_EXTERN ZZFP(ZZH,id) id;* this line generates an instance of the hyper-class in your file which has *#define ZZmain*, and only an *extern* reference in all other files.

Use *#ifdef* and *#endif* around whole macros, not just for sections of macros. For example this works fine:

```
#ifdef UNIX
#define myMacro(a){\
    a=a+3; \
}
#else
#define myMacro(a){\
    a=a+2; \
}
#endif
```

but this is a problem with most compilers:

```
#define myMacro(a){\
#ifdef UNIX
    a=a+3;\
#else
    a=a+2;\
#endif
}
```

16.6 HIERARCHICAL ORGANIZATIONS

The C version of Organized C (see the green manual, also called OrgC) allows the creation of one organization from another through hierarchical macros. For example, a *TRIANGLE* can be derived from a *RING*, and so on. At the beginning, when OrgC worked only with C, this was considered to be an important feature, similar to C++ inheritance. It is interesting that, over two years of industrial use, hardly anybody has used these hierarchical macros. Perhaps the data objects for basic data organizations are simple enough to be treated individually. Perhaps, the flat model is easier to maintain. Or, perhaps, run-time performance, which is the ultimate objective for a library like this, is easier to tune without hierarchy.

If you want to create organizations hierarchically in C++, we suggest that you use the inheritance mechanism. If you are just curious about the original hierarchical macros, look at Chap. 17.6 in the Users Guide for "Data Manager for C".

« Chapter 15: Reporting Errors

Chapter 17: Merging and Reducing Libraries »

17. MERGING AND REDUCING LIBRARIES

If you start to develop your own library of macros and functions, you have to resolve several problems:

1. If you update the *zzmaster* file and you receive a new version of OrgC later, the *zzmaster* will collide with the one you have already developed.
2. If both you and your colleague developed new macros, how do you merge the two libraries?
3. If you only need several organizations, it does not make much sense to drag along hundreds of macros. You may want to reduce the library to only those macros which you will actually use.

The best strategy for developing your own library is not to modify the *zzmaster* file, but to create your own file under a different name, say, *myMaster*. The *myMaster* file has the same format as the *zzmaster* file but contains only the organizations you developed yourself. All the files that contain your new macros and parametric functions will be stored together with OrgC files in the *orgC/macro* directory.

Merging Libraries:

When you are ready to merge the original OrgC library with your new library, rename the original *zzmaster* to something else, say *qqmaster*. Then run the program that merges the two libraries and create a combined file called *zzmaster*:

```
cd orgC/macroÄ
zzmerge qqmaster myMaster zzmaster
```

This can be repeated several times, gradually merging more libraries:

```
zzmerge master1 master2 aMaster
zzmerge aMaster master3 bMaster
zzmerge bMaster master4 zzmaster
```

Reducing Libraries:

If you want to select only a limited set of organizations and functions, make a copy of *zzmaster*, edit it, and delete the organizations and functions you don't want. Don't worry about indices and pointers, and change only two sections: *ZZorganization*{ and *ZZfunction {*.

Be careful that you don't delete too many organizations. You need the organization for every function you keep.

Then run *zzselect*, which creates a new, reduced *zzmaster*, properly rearranged with all indices and pointers. The total run may look like this:

```
cd orgC/macro
```

cp zzmaster qqmaster
cp zzmaster rrmaster
vi rrmaster ... delete some organizations and functions
zzselect qqmaster rrmaster zzmaster

Chapter 16: Adding New Features
Chapter 18: Multi User Mode

18. MULTI USER MODE

OrgC++ may be used simultaneously at several levels for more complicated projects. The rules are simple:

1. Each level controls its own objects. *ZZ_HYPER_* declarations must be in the same file where their objects are declared. Exceptions: *GENERAL_LINK* and *SINGLE_LINK* may point across levels.
2. Subprojects may use the objects and organizations of the projects above them.
3. For each subproject, the class generator must run on a file that combines all *ZZ_*statements from that level up.
4. SELF_ID can be declared only at the highest level.

There are three basic styles of using OrgC++ on large projects.

Chap. 18.1describes the situation where OrgC++ is used to design and control a central database used by many programmers, who use OrgC++ calls to access the database, but not for their private data.

Chap. 18.2 shows a general case, where OrgC++ is used for the central database and for private data by individual programmers.

Chap. 18.3 deals with several independent OrgC++ projects that must co-exist without interference under a large project.

18.1 Central Database

Let us assume that we want to use OrgC++ to design and control a central database. Peggy, who is the database manager, will keep class declarations in a special include file called, for example, *database.h*.

We assume that Peggy will keep all database files in the *database* directory; nobody except Peggy will have write access to that directory.

After declaring the data and its organization through *ZZ_HYPER_* statements, Peggy will run the class generator: *orgC/zzprep database.h*. That will create the *database/ZZinclude.h* file.

All programmers using the database will include in their programs:

```
#include "database/ZZinclude.h"
#include "database/database.h"
```

They will be able to access the central database through OrgC++ calls without even calling the class generator.

Peggy will have to run the class generator any time that the data definition or organization has changed.

18.2 Several Levels of Data

The *orgC/test/multi* directory contains an example, which demonstrates how to handle a multilevel project in a variety of situations. A script to run this test is included as a part of the standard regression test, see *test/cregr* for the SUN C++ compiler, or *test\b4pregr.bat* for Borland C++.

The project involves the following people and directories:

project leader,	directory: main,	files: main.c proj.h
John,	directory: jDir,	files: j1.c j2.c
Susan,	directory: sDir,	files: s1.c s2.c s3.c sIncl.h
Peter,	directory: pDir,	files: p.c

The definition of objects (structures) common to the whole project is in the *proj.h* file which also contains all *ZZ_EXT_*... and *ZZ_HYPER_*... statements common to the whole project.

John is not using any additional organization of data; his programs operate only on common data.

In addition to common data, Susan sets up her own temporary organization of data. The additional *ZZ_EXT_*... and *ZZ_HYPER_*... references are in her *sIncl.h* file.

Peter also uses some additional data structures, but since all his programs are in one file, he does not use a separate header file. All his *ZZ*... references are in the *p.c* file.

The compilation and linking sequence is:

1. Project leader runs *zzprep* on *proj.h*. This creates *ZZinclude.h* in the *main* directory.
2. Now all three programmers can develop and debug their programs independently. John just uses *main/ZZinclude.h*, because he works with global data only. Susan combines *proj.h* and *sIncl.h* and runs them through *zzprep*. That creates her own *ZZinclude.h*, which covers both global data and her own. Peter does the same thing, except that his case is simpler because he has only one file.
3. When the three programmers have their programs ready, they create library files, one in each directory.
4. The project leader compiles *main.c*, and links the whole project together. She must also link it to the OrgC library.

18.3 Co-existing independent projects

Sometimes it happens that, inside a large project such as a telephone switching system or a CAD system for VLSI chips, several programmers want to use the OrgC++ library for their part of the project, while individual data sets are completely independent.

An example of such a situation is the implementation of the *PAGER* in the OrgC++ library. It is implemented with the OrgC library [it has its own private data in file *lib/pager.hpp*], but when you link your application to the library, you don't even know about it.

One possibility is to use *ZZ_LOCAL_*.. instead of *ZZ_HYPER_*.. declarations, and hide the entire implementation inside a special class which encapsulates the whole subproject (see Chap.8.5 on how to use *ZZ_LOCAL_*..).

The second possibility is to use *#define ZZ_LOCAL* instead of *#define ZZmain.* The names are similar, but there is a vast difference between

```
#define ZZ_LOCAL
```

and

```
class Employee;
ZZ_LOCAL(myRing,Employee,Employee);
```

#define ZZ_LOCAL makes all OrgC++ internal type tables and global control variables static. This is the method used internally in our library to implement the pager. The current drawback is (and I believe this is only a temporary limitation) that such a subproject cannot save/retrieve data from disk.

« Chapter 17: Merging and Reducing Libraries

Chapter 19: Revision History »

19. REVISION HISTORY

For various reasons, you may want to know the differences between individual versions of the software. Anybody who is running on a version older than 4.8 should request an update. Backward compatibility for data files remained all through the time period recorded here:

VERSION 5.0

Corrections:

msftregr.bat, msftchk.bat expanded by test23f (memory blasting). There was no program with memory blasting in the regression test for VC++.

Improvements:

Function del() for the HASH organization has been improved so that it detects situations when an object seems to be in a hash table, but when you ask for it, it is not there.

The performance of the *PAGER* has been improved by keeping the track of dirty pages, and flushing only dirty pages to disk when *close()* is called. The operation is also faster due to the use of *memcpy()* internally when copying large blocks of bytes.

array.c is new, ZZhashStr does not involve other function calls.

hyppage is new, multiple calls to pager.close() are now permitted without a crash

del() for ZZ_HYPER_HASH prints error messages if the function cannot find the object. Changes in delhash, msgs.c.

New Features:

Function *newFun()* allows you to switch between different hashing functions while running your program.

An ARRAY can be treated as a *sorted collection*, adding and removing items while maintaining the array ordered (sorted).

A new function *etOrd()*, performs binary search on the sorted array.

New functions available for *PAGER*: *fill()* returns the highest byte address currently used by the *PAGER*; *flush()* flushes all dirty pages to the disk.

Unresolved issues:

Under memory blasting, open() does not return t[], only v[] !!! Testing for t[] seems to give an error even it there isn't any.

VERSION 5.1

Date: Dec.16/98

Corrections:

zzincl.h used to generate warnings in the ZZinhList section. zzprep.c has been modified to generate better zzincl.h.

The order of constructors for (ZZZiClass) in zzfunc.c used to generate warnings under LINUX - zzprep.c has been modified to correct this.

VERSION 5.4

Date: Nov.16/00

Corrections:

In previous version, under some conditions, util.save() crashed when the REFERENCE organization was used. This problem has been corrected.

VERSION 5.4.1

Date: Jan.17/01

Corrections:

No changes in the source, small changes in the batch/script/include files to overcome NT problem with the 'copy' command, which does not copy the newLine character at the end of the file.

Improvements:

Added test58 to orgc/test. It demonstrates splitting the source to *.h and *.cpp files for individual classes, as usual today.

Significant improvement of the documentation part describing save/open commands (Chap.13).

The problem which was reported as *unresolved issue* in Ver.5.0 - open() not returning t[] properly, has been corrected since but not recorded here. We have verified that it works now.

APPENDIX A: CURRENT ORGANIZATIONS AND OPERATIONS IN C AND C++

ZZ_ORG_SINGLE_RING(id,type); ZZ_HYPER_SINGLE_RING(id,type);

```
type *entry,*p;
int (cmpF*)(const void*,const void*);
```

ZZ_FORWARD(id,p,next);	*next=id.fwd(p);*
ZZ_ADD(id,entry,p);	*entry=id.add(entry,p);*
ZZ_DELETE(id,entry,p);	*entry=id.del(entry,p);*
ZZ_MERGE(id,s,t,NULL);	*s=id.merge(s,t);*
ZZ_SORT(id,cmpFun,entry);	*entry=id.sort(cmpFun,entry);*
ZZ_A_TRAVERSE(id,entry,p){ *...* *}ZZ_A_END;*	*id_iterator it(entry);* *while(p= ++it){* *...* *};* *it.start(entry);*

ZZ_ORG_DOUBLE_RING(id,type); ZZ_HYPER_DOUBLE_RING(id,type);

```
type *entry,*p,*next;
int (cmpF*)(const void*,const void*);
```

ZZ_FORWARD(id,p,next);	*next=id.fwd(p);*
ZZ_BACKWARD(id,p,next);	*next=id.bwd(p);*
ZZ_ADD(id,entry,p);	*entry=id.add(entry,p);*
ZZ_INSERT(id,entry,p);	
	entry=id.ins(entry,p);
ZZ_DELETE(id,entry,p);	*entry=id.del(entry,p);*
ZZ_MERGE(id,s,t,NULL);	*s=id.merge(s,t);*
ZZ_SORT(id,cmpFun,entry);	*entry=id.sort(cmpFun,entry);*
ZZ_A_TRAVERSE(id,entry,p){ *...* *}ZZ_A_END;*	*id_iterator it(entry);* *while(p= ++it){* *...* *};*

ZZ_A_REVERSE(id,entry,p){ *...* *ZZ_A_END;*	*it.start(entry);* *while(p=it- -){* *...* *};*

ZZ_ORG_SINGLE_TRIANGLE (id,parT,chiT); ZZ_HYPER_SINGLE_TRIANGLE(id,parT,chiT);

---

*parT *p,*p1,*p2;*
*chiT *c,*s;*
int (cmpF)(const void*,const void*);*

ZZ_PARENT(id,c,p);	*p=id.par(c);*
ZZ_CHILD(id,c,p);	*p=id.child(c);*
ZZ_FORWARD(id,c,s);	*s=id.fwd(c);*
ZZ_SET(id,p,c);	*id.set(c);*
ZZ_ADD(id,p,c);	*id.add(p,c);*
ZZ_DELETE(id,p,c);	*id.del(p,c);*
ZZ_MERGE(id,c,s,p);	*id.merge(c,s,p);*
ZZ_SORT(id,cmpFun,p);	*id.sort(cmpFun,p);* *id.switchParents(p1,p2);*
ZZ_A_TRAVERSE(id,p,c) *...* *}ZZ_A_END;*	*{id_iterator it(p);* *while(c= ++it){* *...* *};* *it.start(p);*

ZZ_ORG_SINGLE_COLLECT(id,parT,chiT); ZZ_HYPER_SINGLE_COLLECT(id,parT,chiT);

---

*parT *p,*p1,*p2;*
*chiT *c,*s;*
int (cmpF)(const void*,const void*);*

ZZ_CHILD(id,c,p);	*p=id.child(c);*
ZZ_FORWARD(id,c,s);	*s=id.fwd(c);*
ZZ_SET(id,p,c);	*id.set(p,c);*
ZZ_ADD(id,p,c);	*id.add(p,c);*
ZZ_DELETE(id,p,c);	*id.del(p,c);*
ZZ_MERGE(id,c,s,p);	*id.merge(c,s,p);*

ZZ_SORT(id,cmpFun,p);	*id.sort(cmpFun,p);* *id.switchParents(p1,p2);*
ZZ_A_TRAVERSE(id,p,c){ *...* *}ZZ_A_END;*	*id_iterator it(p);* *while(c= ++it){* *...* *};* *it.start(p);*

ZZ_ORG_DOUBLE_COLLECT(id,parT,chiT); ZZ_HYPER_DOUBLE_COLLECT(id,parT,chiT);

---

*parT *p,*p1,*p2;*
*chiT *c,*s;*
int (cmpF)(const void*,const void*);*

ZZ_CHILD(id,c,p);	*p=id.child(c);*
ZZ_FORWARD(id,c,s);	*s=id.fwd(c);*
ZZ_BACKWARD(id,c,s);	*s=id.bwd(c);*
ZZ_SET(id,p,c);	*id.set(p,c);*
ZZ_ADD(id,p,c);	*id.add(p,c);*
ZZ_INSERT(id,c,s);	*id.ins(c,s);*
ZZ_DELETE(id,p,c);	*id.del(p,c);*
ZZ_MERGE(id,c,s,p);	*id.merge(c,s,p);*
ZZ_SORT(id,cmpFun,p);	*id.sort(cmpFun,p);* *id.switchParents(p1,p2);*
ZZ_A_TRAVERSE(id,p,c){ *...* *}ZZ_A_END;*	*id_iterator it(p);* *while(c= ++it){* *...* *};* *it.start(p);*
ZZ_A_REVERSE(id,p,c){ *...* *}ZZ_A_END;*	*it.start(p);* *while(c=it++){* *...* *};* *it.start(p);*

ZZ_ORG_NAME(id,type); ZZ_HYPER_NAME(id,type);

---

*type *p;*
*char *n;*

ZZ_FORWARD(id,p,n);	*n=id.fwd(p);*

ZZ_ADD(id,p,n);	*id.add(p,n);*
ZZ_DELETE(id,p,n);	*n=id.del(p);*

ZZ_ORG_SINGLE_LINK(id,sourT,targT); ZZ_HYPER_SINGLE_LINK(id,sourT,targT);

---

*sourT *s;*
*targT *t;*

ZZ_FORWARD(id,s,t);	*t=id.fwd(p);*
ZZ_ADD(id,s,t);	*id.add(p,t);*
ZZ_DELETE(id,s,t);	*t=id.del(p);*

ZZ_ORG_LIFO(id,type); ZZ_HYPER_LIFO(id,type);

---

*type *p; /* an object */*
*type *entry; /* entry into the stack */*

ZZ_PUSH(id,entry,t);	*id.push(p);*
ZZ_POP(id,entry,t);	*p=id.pop();*

ZZ_ORG_FIFO(id,type); ZZ_HYPER_FIFO(id,type);

---

*type *p; /* an object */*
*type *entry; /* entry into the stack */*

ZZ_PUSH(id,entry,t);	*id.push(p);*
ZZ_POP(id,entry,t);	*p=id.pop();*

ZZ_ORG_DOUBLE_LINK(id,sourT,targT); ZZ_HYPER_DOUBLE_LINK(id,sourT,targT);

---

*sourT *s;*
*targT *t;*

ZZ_FORWARD(id,s,t);	*t=id.fwd(p);*
ZZ_BACKWARD(id,t,s);	*s=id.bwd(t);*
ZZ_ADD(id,s,t);	*id.add(p,t);*
ZZ_DELETE(id,s,t);	*t=id.del(p);*

ZZ_ORG_GENERAL_LINK(id,sourT); ZZ_HYPER_GENERAL_LINK(id,sourT);

---

*sourT *s;*
*char *t;*

ZZ_FORWARD(id,s,t);	*t=id.fwd(p);*
ZZ_ADD(id,s,t);	*id.add(p,t);*
ZZ_DELETE(id,s,t);	*t=id.del(p);*

ZZ_ORG_REFERENCE(id,sourT,targT); ZZ_HYPER_REFERENCE(id,sourT,targT);

---

*sourT *s; targT *t; int count;*

ZZ_FORWARD(id,s,t);	*t=id.fwd(p);*
ZZ_ADD(id,s,t);	*id.add(p,t);*
ZZ_DELETE(id,s,t);	*t=id.del(p);*
ZZ_SET_REFERENCE(id,t,count);	*id.set(t,count);*
ZZ_GET_REFERENCE(id,t,count);	*count=id.get(t);*

ZZ_ORG_SINGLE_TREE(id,obj); ZZ_HYPER_SINGLE_TREE(id,obj);

---

*obj *p,*c,*s;*

ZZ_PARENT(id,c,p);	*p=id.par(c);*
ZZ_CHILD(id,p,c);	*c=id.child(p);*
ZZ_FORWARD(id,c,s);	*s=id.fwd(c);*
ZZ_SET(id,p,c);	*id.bwd(p,c);*
ZZ_ADD(id,p,c);	*id.add(p,c);*
ZZ_APPEND(id,c,s);	*id.app(c,s);*
ZZ_DELETE(id,NULL,c);<	*id.del(c);*
ZZ_A_TRAVERSE(id,p,c){ *...* *}ZZ_A_END;*	*id_iterator it(p);* *while(c= ++it){* *...* *};* *it.start(p);*

ZZ_ORG_DOUBLE_TREE(id,obj); ZZ_HYPER_DOUBLE_TREE(id,obj);

---

*obj *p,*c,*s;*

ZZ_PARENT(id,c,p);	*p=id.par(c);*
ZZ_CHILD(id,p,c);	*c=id.child(p);*
ZZ_FORWARD(id,c,s);	*s=id.fwd(c);*
ZZ_BACKWARD(id,c,s);	*s=id.bwd(c);*
ZZ_SET(id,p,c);	*id.bwd(p,c);*
ZZ_ADD(id,p,c);	*id.add(p,c);*
ZZ_INSERT(id,c,s)	*id.ins(c,s);*
ZZ_APPEND(id,c,s)	*id.app(c,s);*
ZZ_DELETE(id,NULL,c);	*id.del(c);*
ZZ_A_TRAVERSE(id,p,c){ *...* *}ZZ_A_END;*	*id_iterator it(p);* *while(c= ++it){* *...* *};*
ZZ_A_RETRACE(id,p,c){ *...* *}ZZ_A_END;*	*it.start(p);* *while(c=it- -){* *...* *};*

ZZ_ORG_SINGLE_GRAPH(id,vertex,edge); ZZ_HYPER_SINGLE_GRAPH(id,vertex,edge);

---

*vertex *v,*t[2];*
*edge *e,*s;*

ZZ_ADD(id,t,e);	*id.add(t,e);*
ZZ_DELETE(id,t,e);	*id.del(e);*
ZZ_NODES(id,t,e);	*id.nodes(t,e);* *e=id.edge(t);* *s=id.fwd(e);* *id.set(v,e);*
ZZ_A_TRAVERSE(id,t,e){ *...* *}ZZ_A_END;*	*id_iterator it(t[0]);* *while(e= ++it){* *t[1]=it.adj();* *...* *};* *it.start(t[0]);*

ZZ_ORG_DIRECT_GRAPH(id,vertex,edge); ZZ_HYPER_DIRECT_GRAPH(id,vertex,edge);

```
vertex *v,*t[2];
edge *e,*s;
```

ZZ_ADD(id,t,e);	*id.add(t,e);*
ZZ_DELETE(id,t,e);	*id.del(t,e);*
ZZ_NODES(id,t,e);	*id.nodes(t,e);* *e=id.edge(t);* *s=id.fwd(e);* *id.set(v,e);*
ZZ_A_TRAVERSE(id,t,e){ *...* *}ZZ_A_END;*	*id_iterator it(t[0]);* *while(e= ++it){* *inset pls t[1]=it.adj();* *...* *};* *it.start(t[0]);*

ZZ_ORG_1_TO_1(id,srs,rel,targ); ZZ_HYPER_1_TO_1(id,src,rel,targ);

ZZ_ORG_1_TO_N(id,src,rel,targ); ZZ_HYPER_1_TO_N(id,src,rel,targ);

ZZ_ORG_M_TO_1(id,src,rel,targ); ZZ_HYPER_M_TO_1(id,src,rel,targ);

ZZ_ORG_M_TO_N(id,src,rel,targ); ZZ_HYPER_M_TO_N(id,src,rel,targ);

```
Source *s; Relation *r; Target *t;
```

ZZ_ADD_RELATION(id,s,r,t);	*id.add(s,r,t);*
ZZ_DELETE_RELATION(id,r);	*id.del(r);*
ZZ_FORWARD(id,s,r);	*r=id.fwd(s);*
ZZ_BACKWARD(id,t,r);	*r=id.bwd(t);*
ZZ_SOURCE(id,s,r);	*s=id.source(r);*
ZZ_TARGET(id,t,r);	*t=id.target(r);*
ZZ_A_TRAVERSE(id,s,r){ *...* *}ZZ_A_END;*	*id_sIterator it(s);* *while(r= ++it){ ... }* *id.start(s);*

ZZ_A_RETRACE(id,t,r){ *...* *}ZZ_A_END;*	*id_tIterator it(t);* *while(r= ++it){ ... }* *id.start(t);*

ZZ_ORG_ARRAY(id,holder,obj); ZZ_HYPER_ARRAY(id,holder,obj);

```
int sz,incr,wMark;                      /* size, increment, waterMark */
int i;                                  /* index */
int (cmpF*)(const void*,const void*);   /* function to compare two objects */
void (bck*)(void *,int);                /* callback function */
holder *hp;
obj *a;                                 /* start of the array */
obj *e;                                 /* pointer to an object/array */
obj r;                                  /* element of the array */
void *vp;                               /* pointer to be NULL or undetermined value*/
```

ZZ_FORM_ARRAY(id,hp,sz,incr,a);	*a=id.form(hp,sz,incr);* *vp=id.formed(hp);*
ZZ_FREE_ARRAY(id,hp);	*id.free(hp);*
ZZ_SIZE_ARRAY(id,hp,sz,wMark,incr);	*sz=id.size(h•p,&wMark,&incr);*
ZZ_INDEX_ARRAY(id,hp,i,e);	*e=id.ind(hp,i);*
ZZ_HEAD_ARRAY(id,hp,e);	*e=id.head(hp);*
ZZ_RESET_ARRAY(id,hp,wMark,incr);	*id.reset(hp,wMark,incr);*
ZZ_PUSH(id,hp,e);	*id.push(hp,e);*
ZZ_POP(id,hp,e);	*e=id.pop(hp);*
ZZ_SORT(id,cmpF,hp);	*id.sort(cmpF,hp);*
ZZ_IN_HEAP(id,cmpF,hp,e,bck);	*id.inHeap(cmpF,hp,e,bck);*
ZZ_OUT_HEAP(id,cmpF,hp,&r,bck);	*id.outHeap(cmpF,hp,&r,bck);*
ZZ_UPDATE_HEAP(id,cmpF,hp,i,bck);	*id.updHeap(cmpF,hp,i,bck);*
ZZ_DELETE_HEAP(id,cmpF,hp,i,bck);	*id.delHeap(cmpF,hp,i,bck);*

ZZ_HYPER_HASH(id,holder,obj); ZZ_HYPER_HASH(id,holder,obj);

```
int sz,num,slot;        /* size, number, slot */
holder *hp;             /* holder object */
obj *p,*s;              /* objects */
obj *t;                 /* object template with a search key */
void *vp;               /* pointer to be NULL or undetermined value*/
```

ZZ_FORM_HASH(id,hp,sz);	*id.form(hp,sz);* *id.formed(hp);*
ZZ_RESIZE_HASH(id,hp,sz);	*id.resize(hp,sz);*

ZZ_ADD(id,hp,p);	*id.add(hp,p);*
ZZ_DELETE(id,hp,p);	*id.del(hp,p);*
*ZZ_SIZE_HASH(id,hp,sz,*num);*	*sz=id.size(hp,&num);*
ZZ_SLOT_HASH(id,hp,slot,p);	*p=id.slot(hp,slot);*
ZZ_FREE_HASH(id,hp);	*id.free(hp);*
ZZ_GET_HASH(id,hp,t,p);	*p=id.get(h,t);* *id.newFun(h);*
ZZ_A_TRAVERSE(id,s,p){ *...* *}ZZ_A_END;* *it.start(s);*	*id_iterator it(s);* *while(p= ++it){* *...* *};*

ZZ_ORG_TIME_STAMP(obj); ZZ_HYPER_TIME_STAMP(obj);

---

char ts[6]; / time record, byte encoded numbers: Y M D H M S */*
*obj *p,*s; /* two objects */*

ZZ_GET_TIME_STAMP(p,ts);	*p->getTime(ts);*
ZZ_SET_TIME(p);	*p->setTime();*
ZZ_CMP_TIME_STAMP(p,s,i);	*i=p->cmpTime(s);*

ZZ_ORG_PROPERTY(obj); ZZ_HYPER_PROPERTY(obj);

```
TEXT prop;      /* property type, for example: int */
char *pTyp;     /* property type as string, for example:"int"*/
obj *p;         /* given object */
char *label;    /* property label */
char **val;     /* property value always treated as an array */
int n;          /* size of the array */
FILE *fp;       /* output file for the print */
```

ZZ_SET_PROPERTY(prop,p,label,val,n);	*p->setProp(pTyp,label,val,n);*
ZZ_GET_PROPERTY(pTyp,p,label,val,n);	*val=(char*)p->getProp(label,&pTyp,&n);*
ZZ_DELETE_PROPERTY(p,label);	*p->delProp(label);*
ZZ_A_TRAVERSE_PROPERTY(pTyp,p,label,val,n){ *...* *}ZZ_A_END;*	*obj_propIterator it(p);* *while(val=p->next(&label,&pType,&n)){* *};*

ZZ_ORG_PAGER(id,hp); ZZ_HYPER_PAGER(id,hp);

```
Holder *hp;             /* holder object */
```

```
char *fn;               /* file name */
int ps;                 /* page size */
int np;                 /* number of pages */
int ii;                 /* 0= not initialized, 1= by '\0', 2= by' ' */
long ind;               /* byte address within the file */
char *buff;             /* memory buffer */
int n;                  /* object size */
int mode;               /* 0=read 1=write */
char *ptr;              /* returned pointer to 'ind' location */
```

ZZ_FORM_PAGER(id,hp,fn,ps,np,ii);	*id.form(hp,fn,ps,np,ii);*
ZZ_IO_PAGER(id,hp,ind,buff,n,mode);	*id.io(hp,ind,buff,n,mode);*
ZZ_ADDRESS_PAGER(id,hp,ind,ptr);	*ptr=id.addr(hp,ind);*
ZZ_CLOSE_PAGER(id,hp);	*id.close(hp);* *id.fill(hp);* *id.flush();*

ZZ_HYPER_TYPE(id);

This organization works in C++ only

```
int n;                          // total number of types used
int tp,tt;                      // index into the internal type table
text objType;                   // object type
void *vp;                       // object pointer
void **vv;                      // pointer to a pointer
void *r1,*r2;                   // range limits of the v.f.table
zzTypeInfo inf;                 // type info structure
n=id.num();                     // returns number of types
tp=ZZgetType(objType);          // gets type index
id.trueType(&vp,&tp);           // updates vp and tp to the
true pointer and type id.info(tp,&inf);// returns info for type tp
id.virtRange(&r1,&r2);          // returns the range of v.f.pointers for all types
id_iterate it(tp);              // iterates through the base classes and members of
tp
while((tt= ++it)=0){ ... }
type_pointers ptrs(vp,tp);      // traverse HYPER pointers on vp
while(vv= ++ptrs){ ... }
struct zzTypeInfo {             // structure which keeps the type info
char *name;                     // class name
int size;                       // size of this class
char *mask1;                    // 0-filled object with correct v.f/v.c. pointers
char *mask2;                    // 'F' or 'C' for bytes that start v.f./v.c. pointers
int virt;                       // bin packed:
      // 01=abstr.class,02=virt.base class,04=virt.functions present
};
```

ZZ_ORG_UTILITIES(id); ZZ_HYPER_UTILITIES(id);

```
char *s,*p;             /* strings */
int i,mSz,pSz;          /* int memory size, page size */
int fmt,mode,ver;       /* format, mode, and version for open/save */
char *file;
int fc;                 /*file control */
char *bn;               /* name or address of memory block */
void *hook;             /* hook to private data */
char *obj;              /* general object */
FILE *fp;               /* file to print on */
char *lb,*tp;           /* label and type of the property */
void *val;              /* array of property values */
int n;                  /* size of the property array */
void fun(char *obj,int typeInd,int size,char *priv);
type *r;
```

ZZ_STRING_ALLOC(s,p);	*p=id.strAlloc(s);*
ZZ_STRING_FREE(s);	*id.strFree(s);*
ZZ_PLAIN_ALLOC(type,i,r);	*r=new type[i];*
ZZ_PLAIN_FREE(type,i,r);	*delete(r);*
ZZ_OBJECT_CLEAR();	*id.objClear();*
ZZ_BLOCK_ALLOC(i);	*id.blkAlloc(mSz,pSz);*
ZZ_BLOCK_ACTIVE(bn,mode);	*id.blkActive(bn,mode);*
ZZ_BLOCK_UTIL(bn,&hook,mode);	*it.blkUtil(bn,&hook,mode);*
ZZ_BLOCK_FREE(i);	*id.blkFree(i);*
i=ZZerror	*i=id.error();*
ZZerror=0;	*id.ok();*
ZZ_SAVE(file,n,v,t);	*id.save(file,n,v,t);*
ZZ_OPEN(file,n,v,t);	*id.open(file,n,v,t);*
ZZ_CLEAR(n,v,t);	*id.clear(n,v,t);*
ZZ_BIND_POINTER(s,p);	*p=id.bind(s);*
ZZ_KEEP_TABLES();	*id.keepTbl();*
ZZ_FREE_TABLES();	*id.freeTbl()'*
ZZ_SWEAP_SET(n,v,t);	*id.swpSet(n,v,t);*
ZZ_SWEAP_FUNCTION(n,v,t);	*id.swpFun(fun,priv);*
ZZ_SWEAP_FREE;	*id.swpFree();*
ZZ_MODE_SAVE(fmt,mode,ver,fc);	*id.mode(fmt,mode,ver,fc);*
ZZ_CLOSE_BIND;	*id.close();*
ZZ_TYPE_NAME(i,p);	*p=id.type(i);*
ZZ_PRINT_PROPERTY(fp,tp,lb,val,n);	*prtProp(fp,tp,lb,val,n);*

ZZ_DEB_PRT(r,tp)	*id.debug(r,tp);*

Chapter 19: Revision History

Appendix B: References

APPENDIX B: REFERENCES

This appendix provides a list of publications related to the Code Farms libraries. Reprints of the papers are available from Code Farms for a nominal charge covering printing and postage.

[1] Weiss R.: *Software Manages C Data Structures*, Electronic -Engineering Times, Feb.5, 1990, pp.39-45.

Early, popular article explaining the advantages of our approach.

[2] Soukup J.: *Organized C: A Unified Method of Handling Data in CAD Algorithms and Databases*, 27-th ACM/IEEE Design Automation Conference, 1990, pp.425-430.

Describes the main idea, using three examples: netlist for electrical circuits, classical ER example, and a database for a VLSI layout system. All code examples are in C. This paper was written at the time when the C++ version of the library was not developed to its current advanced level.

[3] Soukup J.: *Selecting a C++ Library*, C++ Report, Jan.1992

This article does not describe our library, but it lists features and priorities that we considered important in our design. It also contains a description of a benchmark suitable for a quick evaluation of a class library.

[4] Soukup J.: *Memory Resident Databases*, C++ Report, Feb.1992

Describes how persistent data, such as that provided by our library, can be used for fast, flexible databases.

[5] Soukup J.: *Beyond Templates*, C++ Report, two parts (April and May 1992)

This is a detailed, theoretical analysis with numerous C++ examples.

It explains the advantages of our approach compared to classical C++ libraries and templates, in both run-time performance, code clarity and ease of maintenance.

[6] Galbiati L.: *Comparing Different Implementations of a VLSI Simulation Database*, submitted to fall ICCAD 92

A user report on evaluation and benchmarks of Code Farms C library, with comparisons to commercial object-oriented databases and a custom-designed database. The performance of the program coded with our library was comparable to the custom-designed database; it needed 5 times less memory and had 20-times faster data access than a leading commercial database.

[7] Hutchings B.L.: *Achieving CAD Data Persistence With C++*,submitted to 1992 OOPSLA Conference

A user report on a difficult case of a large database (250+ classes), which was originally designed without storage to disk. The author selected Code Farms' library to add persistency to the database, and the paper describes his approach and experience.

[8] Soukup J.: *Maze Router Without A Grid Map*, submitted to the fall ICCAD 1992

Decription of a special algorithm finding routes through a maze of complex obstacles. The paper includes a full listing of the program coded with the C version of our library.

[9] Soukup J.: *The Secret Of Efficient Software Design: Internal Data Organization*, Electro Convention, Boston, May 12-14, 1992

Describes the advantage of separating data objects from data relations, and how easily this is done with the Code Farms library.

The paper evolves around a sample problem involving towns connected by highways. The task is to find the fastest route between two given towns. The paper includes a full C++ listing of the program.

[10] Soukup J.: *Taming C++: Pattern Classes and Persistence for Large Projects*, Addison-Wesley, July 1994, ISBN 0-201-52826-6.

The book describes a new approach to the implementation of data structure libraries, and patterns in general. Also, various forms of persistent data are discussed in detail.

[11] Soukup J.: *Implementing Patterns, in Pattern Languages of Program Design* (edited by Complien & Schmidt), Addison-Wesley 1995, pp.395-415, ISBN 0-201-60734-4.

This paper from the the first PLoP conference shows that structural patterns can be implemented in the style which we use in the OrgC++ library.

[12] Soukup J.: *Intrusive Data Structures* (3 part article), C++ Report, May, July, Oct.1998

[13] Soukup J.: Quality Patterns, C++ Report, Oct.1996 also Managing Groups of Cooperating Classes in the same issue.

[14] Vadaparty K.: *Memory Resident Databases* 1997.

« Appendix A: Current Organizations Chapter 1: Introduction »