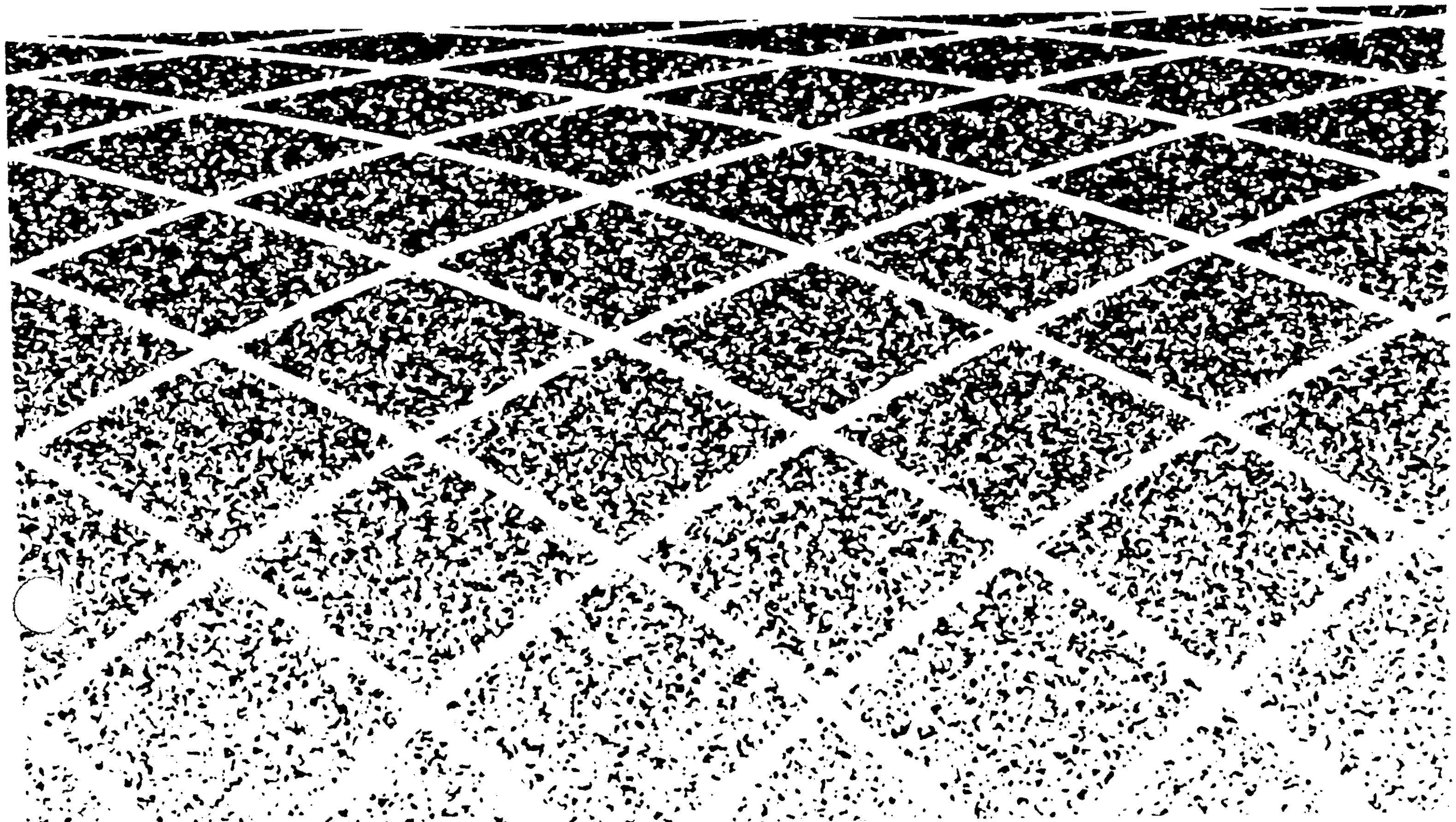
AT&T

UNIX® System V
AT&T C++ Language System
Release 2.0

Release Notes
Select Code 307-090



Contents

Figures and Tables

Preface

Preface

The *AT&T C++ Language System Release Notes* describe Release 2.0 of the AT&T C++ Language System. The manual is part of a set of four documents that are supplied with your C++ Language System. The other documents are:

- the *Product Reference Manual*, which provides a complete definition of the C++ language supported by Release 2.0 of the C++ Language System.
- the *Selected Readings*, which contains papers describing aspects of the C++ language
- the *Library Manual*, which describes the three C++ class libraries and tells you how to use them

The *Release Notes* consist of four chapters and two appendices, which describe how to install and use the translator, changes in the C++ language for this release, and other information you need to know:

- Chapter 1 is a general description of the C++ Language System and new features that are part of this release. You should read this chapter as a general introduction to the release.
- Chapter 2 is a description of the contents of Release 2.0. This chapter includes a diagram of the contents of the tape or diskettes from which you install the C++ Language System. You can use this diagram and the accompanying descriptions as a reference when you install and use the Language System.
- Chapter 3 tells you how to install the C++ Language System and how to port it to machines for which it is not directly supported.
- Chapter 4 covers compatibility with previous releases of the C++ Language System; it describes anything that has changed in Release 2.0 and might require you to make changes in code written for previous releases. You should read this chapter if you have existing code running under a previous release of the C++ Language System.
- Appendix A describes known problems with the C++ Language System and suggests workarounds for these problems.
- Appendix B is a list of documentation and ordering information.

To make the best use of the *Release Notes*, you must be familiar with the C programming language and the C programming environment under the UNIX® operating system. Refer to Appendix B for further sources of information about these topics.

1 Introduction

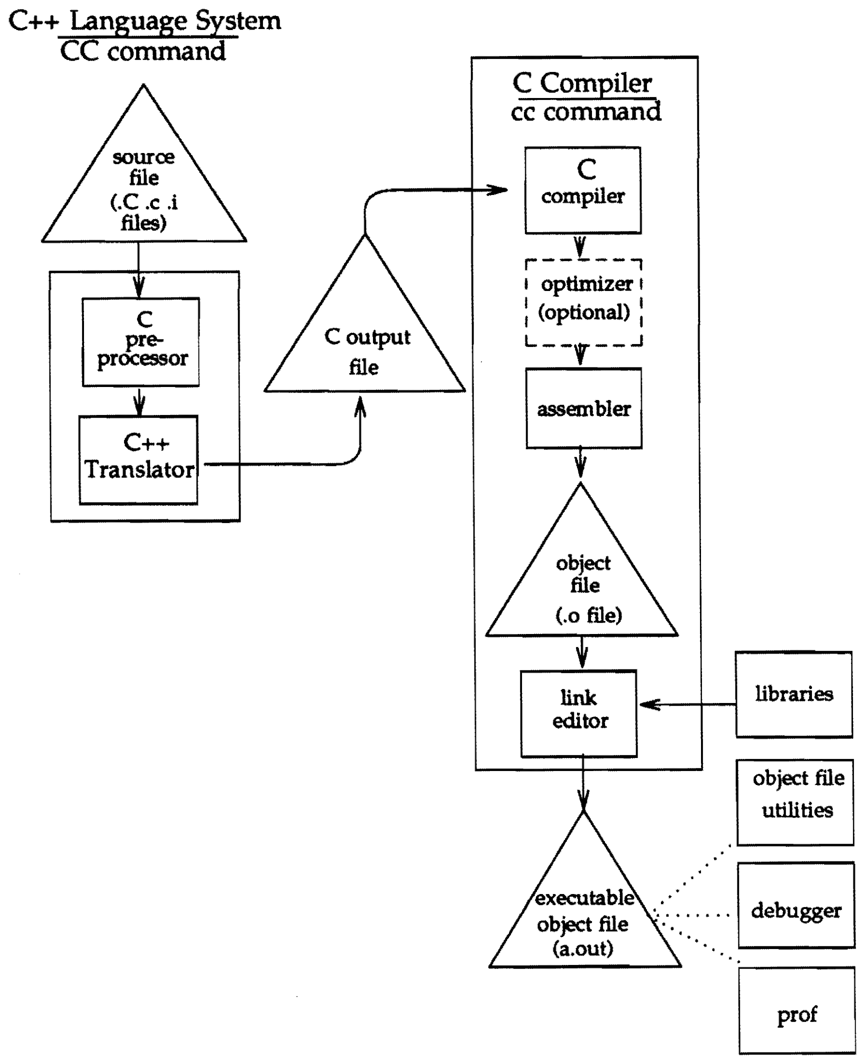
Introduction

The AT&T C++ Language System

The AT&T C++ Language System, Release 2.0, is a translator that translates C++ source code to C source code. It supports the C++ programming language (as described in the enclosed *C++ Language System Product Reference Manual*). The CC command invokes the translator, which does semantic and syntactic checking on the C++ input program and translates the C++ program to C language. The CC command then invokes the C compiler on your machine to compile the resulting C program and run related processes such as linking function libraries.

Figure 1-1 shows the operation of the CC command and the processes it invokes:

Figure 1-1: Operation of the CC Command



The C++ Language System can run on most UNIX systems with a C compiler that supports the following features:

- long variable names (of at least 31 characters)
- structures as arguments to functions and return values from functions

See "Installation Procedures" for detailed prerequisites and information on porting.

New Features

Release 2.0 of the C++ Language System supersedes Release 1.2. This new release supports the following language changes, enhancements, and new features added since Release 1.2 was introduced (see Chapter 1 of the C++ *Language System Selected Readings*).

- multiple inheritance
- type-safe linkage
- **new** and **delete** operators as class members
- overloading of the ->, ->*, and , operators
- **const** and **static** member functions
- static initialization

Additionally, Release 2.0 provides the following enhancements to the translator product:

- enhanced portability
- revised C++ libraries supporting I/O, tasking operations, and complex arithmetic
- simplified installation procedures
- expanded documentation

Hardware

The AT&T C++ Language System Release 2.0 can be installed on the following machines:

- AT&T 3B Series Computers
- DEC VAX line of Computers (including VAX BSD machines)
- SUN 2, SUN 3, and SUN 4 Workstations

The Language System has also been successfully ported to other machines, including:

- AT&T 6386 WGS
- Motorola 68000-based Apollo and HP workstations
- Amdahl UTS Computer
- Intel 80286 large model and 80386-based machines

- Hewlett-Packard 9000 series 800 HP-PA based machines
- IBM RT Personal Computer
- Data General MV Computers running AOS/VS and DG/UX

The C++ Language System can also be ported to other machines not on this list. Porting to machines besides the AT&T 3B series, DEC VAX line of computers, and SUN 2/3/4 workstations requires that you have access to an AT&T 3B series, DEC VAX, or SUN 2/3/4 computer, or that you have access to an existing working C++ translator. For information about porting, see Chapter 3, under "Porting the C++ Translator."

2 Contents of the Release

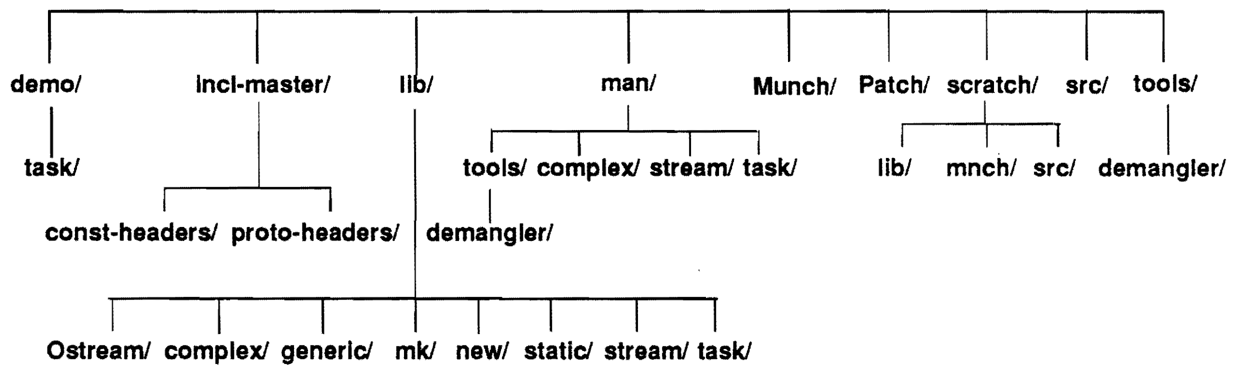
Contents of the Release

The AT&T C++ Language System includes:

- a tape or a set of five floppy diskettes (depending on the machine on which the release will be installed) containing the files that make up the C++ Language System
- the *AT&T C++ Language System Release Notes* (this document)
- the *AT&T C++ Language System Product Reference Manual,* which describes the C++ programming language in detail
- the *AT&T C++ Language System Selected Readings,* which contains papers describing aspects of the C++ programming language
- the *AT&T C++ Language System Library Manual,* which describes the **iostream**, **task**, and **complex** class libraries provided with the C++ Language System

Figure 2-1 shows the structure of the directories that will be created when you extract the tape following the instructions in Chapter 3 of this document.

Figure 2-1: Directory Structure of the AT&T C++ Language System Tape



In the root directory you will find the following files:

- a **README** file, containing up-to-date information about the release
- the program **makefile**, which **make** uses to build the translator
- the program **szal.c**, an aid for porting the translator
- the program **bsd.fix**, used to adapt the translator for users running the 4th Berkeley Software Distribution (BSD)
- the CC command, which you use to invoke the translator and the other tools required to produce an executable C++ program.

The **mnch** subdirectory under the **scratch** directory is only found on tape versions of the C++ Language System.

NOTE Make sure you read the **README** file before attempting to install the C++ Language System.

The following list describes the contents of the subdirectories under the Language System root directory:

src contains the C++ source for the translator itself.

incl contains the **#include** files for use with C++. Many of these files are standard UNIX system header files with argument types added to the function declarations. (Chapter 3 describes how to convert your own C header files by adding argument types and making other changes to function declarations.) The other files in **incl** are headers for C++ library functions.

The **incl** directory is not included on the tape, but is created when you run **make**. The header files that end up in **incl** differ depending on the machine for which you build the translator. The prototypes for the header files are in the **incl-master** directory when you install the translator.

The following list shows the contents of **incl** on an AT&T 3B2 Computer:

Ostream.h	**fstream.h**	**memory.h**	**setjmp.h**	**sys/**
assert.h	**ftw.h**	**mon.h**	**signal.h**	**sysent.h**
common.h	**generic.h**	**new.h**	**stdarg.h**	**task.h**
complex.h	**grp.h**	**nlist.h**	**stddef.h**	**time.h**
ctype.h	**iomanip.h**	**osfcn.h**	**stdio.h**	**unistd.h**
curses.h	**iostream.h**	**plot.h**	**stdiostream.h**	**ustat.h**
dial.h	**ldfcn.h**	**pwd.h**	**stdlib.h**	**utime.h**
dirent.h	**libc.h**	**rand48.h**	**stream.h**	**utmp.h**
errno.h	**malloc.h**	**regcmp.h**	**string.h**	**vector.h**
fcntl.h	**math.h**	**search.h**	**strstream.h**	

incl has one subdirectory, **incl/sys**. This subdirectory contains the UNIX System V headers for files that you may be used to seeing in **/usr/include/sys**.

ipc.h	**msg.h**	**signal.h**	**stat.h**	**times.h**	**utsname.h**
lock.h	**sem.h**	**shm.h**	**statfs.h**	**types.h**	

lib contains the following subdirectories, which have the C++ source for the library functions included with the C++ Language System:

- **Ostream**, the old Release 1.2 C++ I/O library
- **complex**, the complex arithmetic library (see Chapter 1 of the *C++ Language System Library Manual*)
- **generic**, a common error-reporting function and generic stack and vector types
- **mk**, the directory in which the **libC.a** makefile resides. **libC.a** is the standard C++ library, and is composed of functions defined in the files under **generic**, **new**, **static**, and **stream**.

- **new**, which handles run-time memory management
- **static**, which handles invocation of static constructors and destructors (see Chapter 3 of the C++ *Language System Selected Readings* for descriptions of constructors and destructors)
- **stream**, the C++ iostream library (see Chapter 3 of the C++ *Language System Library Manual*)
- **task**, the C++ task library. This package is machine dependent and is currently implemented only on the AT&T 3B line and 6386 WGS computers, the DEC VAX line of computers, and SUN workstations using the Motorola 68000 processor family (though Chapter 2 of the C++ *Language System Library Manual* provides information on how to port it to other machines).

Munch contains the source files necessary to compile the **munch** version of the C++ Translator.

Patch contains the source files necessary to compile the **patch** version of the C++ Translator. See Chapter 3 under "The Patch and Munch Options" for more information.

scratch contains the intermediate C source for the translator, which you need only if you bootstrap the translator on a machine that has no existing C++ translator.

The source diskettes for the AT&T 3B2 Computer do not contain scratch files, since new scratch files can be generated using the binary version.

The **scratch** directory contains additional files: a **makefile**, the file **bsd.sed**, and various other **sed** files, which are used to modify the C source for other machines.

Scratch contains the following subdirectories:

- **lib**, which includes a makefile and the C source for **libC.a**
- **mnch**, the C source for the program **munch**, which is used by the translator to see that static constructors and destructors are called. This subdirectory is only provided on tape versions of the C++ Language System.
- **src**, which includes a makefile and the C source for the translator

3 Installing the Translator

Overview

This chapter explains how to install the C++ translator, which is the principal component of the AT&T C++ Language System, on a computer running the UNIX operating system.

The C++ Language System is available for the machines listed below.

- AT&T 3B2, 3B5, 3B15, and 3B20 Computers
- DEC VAX line of computers
- SUN 2, SUN 3, and SUN 4 Workstations

If you have an existing C++ translator, or access to an AT&T 3B, DEC VAX, or SUN 2/3/4 computer, you can port the translator to many other computers, *including* the:

- AT&T 6386 WGS Computer
- Motorola 68000-based Apollo Workstations
- Amdahl UTS Computers
- Intel 80286 large model and 80386-based machines
- Hewlett-Packard 9000 series 800 HP-PA based machines
- IBM RT Personal Computer
- Data General MV Computers running AOS/VS and DG/UX

The AT&T 3B series computers, DEC VAX line of computers, and SUN 2/3/4 workstations require no porting to build the translator. If you have one of these machines, and you do not have a previous version of the translator running, you can build an intermediate version using the ..c files in the scratch directory on the translator tape, then use this intermediate version to build the final version of the translator.

Other computers cannot use these ..c files to build an intermediate translator, and require porting to build the translator.

Porting to a machine requires that you:

- have an existing version of the translator on any machine

  or

- have access to an AT&T 3B series computer, DEC VAX computer, or SUN 2/3/4 workstation. You can then build the translator directly on the AT&T 3B, VAX, or SUN machine, and use this as your existing translator to port the translator to another machine.

Some work has been done to simplify porting to the target machines listed above, for example the inclusion of size and alignment information for these machines in **src/size.h**. However, you should still go through all steps described under "Porting the C++ Translator" for these machines.

For further information, see "Porting the C++ Translator."

NOTE If you are installing the translator on a computer that runs UNIX System V or the 4th Berkeley Software Distribution (BSD), read the section on "The Patch and Munch Options."

Prerequisites

Software Dependencies

- The instructions in this manual assume your machine runs some form of the UNIX operating system. The C++ translator has been ported to machines running other operating systems, but no guarantees are made as to the success of such porting or support for the translator running under another operating system.
- You need a C compilation system that supports the following features of the C programming language:
    - Flexnames

      The compiler and link editor must be able to handle variable names (including external functions) of at least 31 characters. All characters of these names must be significant. C++ encodes information in function names, which can reach lengths of 100 characters or more in extreme cases. The long names provide for some type checking among program modules.

      If your system accepts variable names of at least 31 characters, but less than 100 characters, see the section "DENSE Makefile Variable."
    - Structure arguments

      Structures must be legal as arguments to functions and as return values from functions. Structure assignments must also be supported.

  Any AT&T 3B Computer running the C Programming Utilities, Issue 4.1 or later, fullfills the requirements for building and running the translator.

Space Requirements

To build the translator, you need three megabytes of free disk storage space.

The translator tape contains about three megabytes of source, as follows:

- scratch directory — about two megabytes
- everything else — about one megabyte

If you already have a version of the C++ translator running on your system, you need only the one megabyte of disk storage for "everything else." However, if you do not have a translator running on your system, you need the two additional megabytes of storage for **scratch** (see "Bootstrapping the Translator").

After you install the translator, you need 500 kilobytes of disk storage space to run it. A breakdown of this requirement follows:

- the translator and other executables — 350 kilobytes
- header files (**/usr/include/CC**) — 150 kilobytes

The Patch and Munch Options

This section describes an optional step that you can choose to take later when building the translator. If you are installing the translator under UNIX System V or BSD, read this section.

By default, **make** builds what is called a **munch** version of the translator. In the **munch** version, the translator has to run the link editor twice when creating an **a.out** to make sure that all static constructors are called before the main program runs.

If you are running either UNIX System V or BSD on your machine, you can build a different version of the translator, called the **patch** version. The patch version provides a utility that manipulates the executable file directly. This replaces munch and lets you avoid having to run the link editor a second time.

If you install the binary version of the translator on an AT&T 3B2 Computer, the patch version is installed by default.

If you build the translator from source, you can decide whether to build a patch version or a munch version (**make** with no arguments creates a munch version by default).

Note that you may not mix components from different versions (e.g., a patch **cfront** and a munch CC), as a translator constituted of such executables will fail.

Instructions for installing a patch version appear in each of the sections on installation.

Installing the Translator on an AT&T 3B2 Computer

If you are installing the translator on an AT&T 3B2 Computer, follow the instructions below. If not, proceed to "Installing the Translator from Tape."

If you ordered the translator for the 3B2 computer, you will receive the following:

- two diskettes labeled *AT&T C++ Language System, Release 2 Version 0.* These diskettes contain the executable files, the header files, and the class libraries for the C++ translator, and are the only diskettes you need to install the translator.
- three diskettes labeled *AT&T C++ Language System Source, Release 2 Version 0.* These diskettes contain the C++ source files for the translator and the C++ libraries. You do not need to install these diskettes to use the translator unless you want to port the translator from your 3B2 to another machine.

NOTE Installing the translator using the following instructions will overwrite any existing translator found in the standard locations on your 3B2. If you wish to save the existing translator, first copy it to another directory before installing the new translator. A list of the files that will be installed (overwriting any previous versions) is given under "Installing Binary C++ Translator on the 3B2 Computer."

Prerequisites

The following packages must be installed on your 3B2 Computer before you can install the C++ translator:

- if you are installing the binary translator, your 3B2 Computer must be running UNIX System V, Release 3 or later
- C Programming Language Utilities, Issue 4.1 or later
- Software Generation Utilities, Issue 4.1 or later

If either of these packages is not installed on your 3B2 Computer, an error message will be displayed when you try to install the translator.

Installing Binary C++ Translator on the 3B2 Computer

You install the translator just like any other utilities package that does not have device drivers. Enter the command:

```
$ sysadm installpkg
```

The system will issue instructions to guide you in installing the translator. When the system asks you to install the removable medium in the diskette drive, put the diskette labeled *AT&T C++ Language System, Release 2 Version 0 — diskette 1 of 2* into the drive and press <CR>.

The files will be copied from the diskette to the hard disk. After the files on each diskette are copied onto the hard disk, **sysadm installpkg** will prompt you to insert the next diskette. Be sure to insert the diskettes in the correct sequence (i.e., 1 of 2, then 2 of 2). These executable files provide a patch version automatically.

Your C++ Language System is now installed. The procedure described above installs the following files and subdirectories in the directories listed:

- CC, the shell script that invokes the C++ translator (in **/usr/bin**)
- **cfront**, the main pass of the translator (in **/usr/bin**)
- **patch**, the secondary pass of the translator (in **/usr/bin**)
- **libC.a**, **libtask.a**, and **libcomplex.a**, the C++ class libraries (in /usr/lib)
- **c++filt**, which decodes the variable name encoding in any error messages (in **/usr/bin**)
- the subdirectory **/usr/include/CC** and its subdirectory **/usr/include/CC/sys**, which contain the C++ header files for your machine

NOTE The remainder of this chapter does not apply to installing the binary translator on the 3B2. You need not read further in this chapter.

Installing Source C++ Translator on the 3B2 Computer

NOTE Install the source on the 3B2 only if you plan to port the translator from your 3B2 to another machine.

If you want to install the source, use the diskettes labeled *AT&T C++ Translator Source, Release 2 Version 0*. Type the command:

```
$ sysadm installpkg
```

The system will issue instructions to guide you in installing the translator. When the system asks you to install the removable medium in the diskette drive, put the diskette labeled *AT&T C++ Translator, Release 2 Source Version 0 — diskette 1 of 3* into the drive and press <CR>. After the files on each diskette are copied onto the hard disk, **sysadm installpkg** will prompt you to insert the next diskette. Be sure to insert the diskettes in the correct sequence (i.e., 1 of 3, 2 of 3, etc.).

All the source files will be copied from the floppies to the hard disk; they can be found under the directory **/usr/src/cmd/CC**.

If you are using your 3B2 as a host machine to port to another machine, continue with the instructions in the next section, starting under "Setting Makefile Variables."

NOTE See the section "Common Problems" at the end of this chapter before beginning to install the translator from source, to read about any problems that apply to the machine on which you want to install the translator.

NOTE Diskettes containing source for the 3B2 do not contain scratch files. You can generate scratch files for porting using the executable installed from the binary diskettes.

Installing the Translator from Tape

NOTE See the section "Common Problems" at the end of this chapter before beginning to install the translator to read about any problems that apply to the machine on which you want to install the translator.

You can install and build the C++ translator directly from tape on:

- AT&T 3B5, 3B15, or 3B20 Computers
- DEC VAX Computers
- SUN 2, SUN 3, and SUN 4 Workstations

You may also port the translator to many other machines. In order to port the translator to another machine, you must either have access to an AT&T 3B series computer, DEC VAX computer, or SUN 2/3/4 workstation, or you must have access to a working version of the translator. Some machines to which the translator has been successfully ported are:

- AT&T 6386 WGS Computer
- Motorola 68000-based Apollo workstations
- Amdahl UTS Computer
- Intel 80286 large model and 80386-based machines
- Hewlett-Packard 9000 series 800 HP-PA based machines
- IBM RT Personal Computer
- Data General MV Computers running AOS/VS and DG/UX

You can port the translator to many other machines not on the above list. If you need to port to other machines, the installation instructions refer you to the section "Porting the C++ Translator" at the appropriate point.

Overview

This section describes the general principles of how you build your Release 2.0 C++ translator.

When you install the Release 2.0 C++ Translator on one of the machines listed above, you are installing source for the translator. This source is itself written in the C++ programming language. Therefore, in order to compile the Release 2.0 translator, you must have a translator on your machine. There are two ways you can provide a translator:

- if you have an existing C++ translator, you can use it to translate the C++ source for the Release 2.0 translator by running the **make** or **make patch** command.
- if you do not have a previous release of the C++ translator, and are installing the translator on an AT&T 3B computer, DEC VAX computer, or SUN 2/3/4 workstation, you can create an intermediate version of the translator using files provided with Release 2.0 and use this intermediate version to build the Release 2.0 translator itself (this is called "bootstrapping" the translator) by making one of the following versions of the translator:

- if you are running UNIX System V or BSD, you can make a patch or munch version using the **make patch** or **make** procedures, respectively
- if you are running another form of the UNIX system, you can make a munch version using the **make** procedure

After you finish building the translator, you can run some procedures to test the operation of the new translator and clean up unneeded files. You can then use your new translator to compile the source for the C++ class libraries.

C++ Translator Roadmap and List of Procedures

Figure 3-1: C++ Translator Installation Roadmap

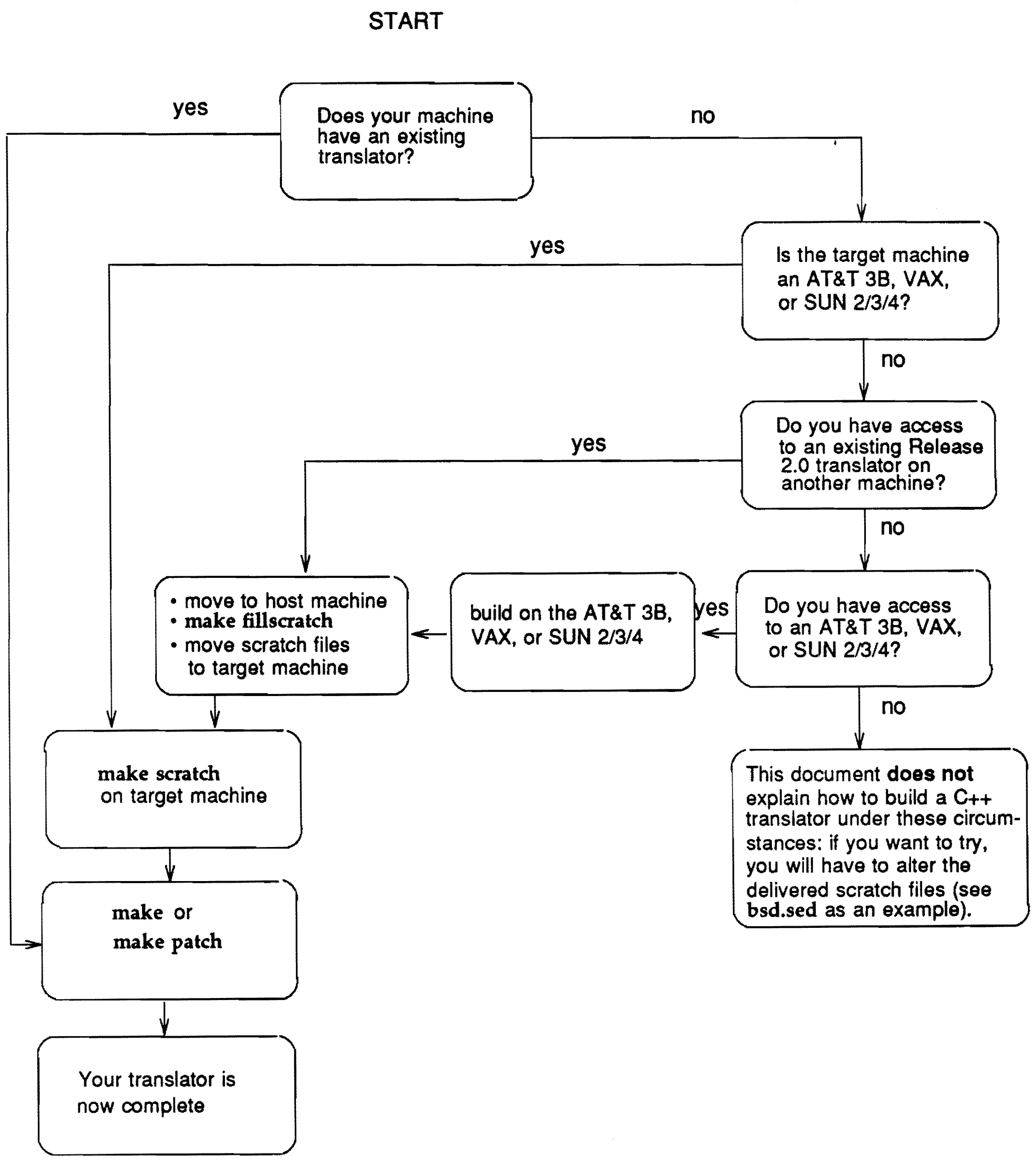

START
Does your machine have an existing translator?
yes
no
Is the target machine an AT&T 3B, VAX, or SUN 2/3/4?
yes
no
Do you have access to an existing Release 2.0 translator on another machine?
yes
no
Do you have access to an AT&T 3B, VAX, or SUN 2/3/4?
yes
no
build on the AT&T 3B, VAX, or SUN 2/3/4
• move to host machine
• make fillscratch
• move scratch files to target machine
make scratch on target machine
make or make patch
Your translator is now complete
This document does not explain how to build a C++ translator under these circumstances: if you want to try, you will have to alter the delivered scratch files (see bsd.sed as an example).

Figure 3-1 is a roadmap showing the various paths through the procedure for building the translator. This roadmap is read from the top down. At each decision point, by answering "yes" or "no" to the question, you can see what procedures you need to do to build the translator.

Figure 3-2 is a list of procedures that guides you through installing the C++ translator from tape. The list of procedures is read from the top down, following the path applicable to your machine (e.g., existing translator, no existing translator, etc.). Note that the procedures at the top and bottom apply to all installations.

NOTE Read the instructions in the following sections thoroughly. The translator installation roadmap and list of procedures are only summaries of how to install the translator. Do not attempt to build the translator using the roadmap or list of procedures.

Figure 3-2: C++ Translator List of Procedures

C++ Translator List of Procedures

- copy files from tape using **cpio** or **tar**
- edit the parameters at the beginning of the **makefile** in the top-level directory to appropriate settings for your machine

with existing translator:	without existing translator:	
build translator: ■ **make** or ■ **make patch**	bootstrap translator: ■ if you are not on an AT&T 3B, VAX, or SUN 2/3/4, port the translator by generating new include files and new intermediate C scratch files using an existing translator on another machine ■ **make scratch** ■ **make install** ■ **make clean** (optional)	
	make a patch version (if UNIX System V or BSD)	make a munch version
	■ **make patch**	■ **make**

on all systems: test, edit, and move files

- **make CC=`pwd`/CC demos**
- **make install**
- set CC in makefile to newly installed CC
- make libraries and install in final locations (optional)
- **make clean** or **make clobber** (optional)
- now that you have a Release 2.0 translator, you can use it to port the translator to other machines

Extracting the Contents of the Tape

The C++ translator tape may be either in **cpio** format or in **tar** format. The format of your tape depends on the machine you specified in your order for the translator. Check your tape; the format is clearly marked on it.

1. Place the tape on the tape drive for your machine.
2. cd to a directory from which you want to install the translator. This will be the working directory from which you do all the installation procedures. When you extract the contents of the tape, they are placed in a directory tree below this working directory.
    1. If the file is in **cpio** format, run the **cpio** command:

    ```
    $ cpio -icvud < tape_drive_spec
    ```

    2. If the file is in **tar** format, run the **tar** command:

    ```
    $ tar xf tape_drive_spec
    ```

NOTE *tape_drive_spec* is the device name of your tape drive, and may differ depending on your machine. For example, on an AT&T 3B15 computer, the device name of tape drive 0 will be **/dev/rmt/0m**. You may also require different options to **cpio** or **tar**, depending on the kind of machine you use; check your operating system user's manual for details.

The contents of the tape will be installed under the current directory. Figure 2-1 in Chapter 2 shows the structure of the directory tree that will be created when you extract the tape.

Setting Makefile Variables

Before building the C++ translator, you must set several variables in the **makefile** located in the top directory installed from the translator tape. This section lists each of these variables, explains its function, and lists the possible settings. You may not need to change some of the settings and some may not apply to your system or machine.

SYS Makefile Variable

The SYS parameter specifies the operating system under which the translator will run. Possible settings for the SYS variable are:

Setting	Operating System
SYSV	UNIX System V
BSD	4th Berkeley Software Distribution

If you set **SYS** to "BSD," the makefile runs the script **bsd.sed** on the files in the **scratch** directory, altering them to match the correct BSD structures.

OS Makefile Variable

The setting of the **OS** parameter determines which set of header files is generated for use with C++. Different machines require different header files since their underlying C compilation systems differ.

The settings for **OS** depend on the setting of **SYS**. The following table shows what settings are available for **OS**, depending on your **SYS** setting:

SYS Setting	OS Setting	Operating System
SYSV	svr2	UNIX System V Release 2
	svr3	UNIX System V Release 3
BSD	bsd2	Berkeley Software Distribution Release 4.2
	bsd3	Berkeley Software Distribution Release 4.3
	sun	SUN operating system
	hpux	Hewlett-Packard UX operating system

Set **OS** to the value that is closest to your machine's operating system. If you are building the translator on a machine not on this list, see "Porting the Translator" for more information.

NOTE When you set **OS** to "sun," the **makefile** determines whether you are running SUNOS 3 or SUNOS 4 automatically. You can also explicitly set **OS** to "sunos3" or "sunos4" to build on a machine running SUNOS 3 or SUNOS 4, respectively.

STDINCL Makefile Variable

The **STDINCL** makefile variable specifies the location of the standard C language **#include** files on your machine. For example, if your **#include** files are in **/usr/include**, you would set **STDINCL** as follows:

```
    STDINCL=/usr/include
```

SYMBOLICLINK Makefile Variable

Set the **SYMBOLICLINK** makefile variable to "1" if you are not running UNIX System V and you want your C++ header files to use symbolic links.

NOTE SYMBOLICLINK should not be set when you are porting the translator.

MACH Makefile Variable

The setting of the **MACH** parameter identifies your machine for building the task library. For example, to build the task library on an AT&T 3B Computer, you would set **MACH** as follows:

```
MACH=3b
```

The following settings are available for **MACH**.

Setting	Machine
3b	AT&T 3B Series Computers (except 3B1)
386	AT&T 6386 WGS and other Intel 80386-based machines
vax	DEC VAX Computers
68k	Motorola 68000-based SUN−2/3 workstations

You do not need to set **MACH** if you do not intend to build the task library. Instructions for building the task library are provided under "Building the Task Library." If you are building the translator on a machine not on this list, see Chapter 2 of the *AT&T C++ Language System Library Manual*, under "Porting the Task Library," for more information.

INSTALL_BIN Makefile Variable

The INSTALL_BIN parameter specifies the directory where you want the translator executables **CC**, **cfront**, **c++filt**, and **patch** or **munch** to reside. For example, if you want them to reside in **/usr/add-on/C++/bin**, set INSTALL_BIN as follows:

```
INSTALL_BIN=/usr/add-on/C++/bin
```

The default value for INSTALL_BIN is **/usr/bin.**

INSTALL_LIB Makefile Variable

Set the INSTALL_LIB parameter in the makefile to the directory where you want the C++ class libraries to reside. For example, if you want them to reside in **/usr/add-on/C++/lib**, set INSTALL_LIB as follows:

```
INSTALL_LIB=/usr/add-on/C++/lib
```

The default value for INSTALL_LIB is **/usr/lib.**

INSTALL_INC Makefile Variable

Set the INSTALL_INC parameter in the makefile to the directory where you want the C++ header files to reside. For example, if you want them to reside in **/usr/add-on/C++/include**, set INSTALL_INC as follows:

```
INSTALL_INC=/usr/add-on/C++/include
```

The default value for INSTALL_INC is **/usr/include/CC.**

NOTE The C++ header files have the same names as the standard C header files, and thus must reside in a different directory from the standard C language **include** directory.

INSTALL Makefile Variable

The **INSTALL** parameter identifies the command used to move files to the installation directory. The default value is "cp" (the UNIX System command **cp**, which copies files). You may substitute a system-specific installation command for this value. You might also substitute the UNIX system **mv** command to move, rather than copy, the files.

CPIO Makefile Variable

The **CPIO** makefile variable specifies the command used to copy a directory tree of newly built header files from the **incl-master** directory to the top-level **incl** directory.

CC Makefile Variable

The CC parameter identifies the C++ translator command to be used to build the translator itself. The default is "CC."

If you are installing the translator on a machine with a translator already present, the **CC** parameter in the top level makefile should point to the location of the **CC** command for your existing translator.

CCFLAGS Makefile Variable

The **CCFLAGS** parameter lists any special options to pass to the **CC** command. This variable defaults to the value "-O -D$(SYS)" (the optimizer and operating system setting).

The default setting for **CPIO** is **cpio -pdlm**, the UNIX System **cpio(1)** command. If your system does not have the **cpio** command, the translator has an alternative shell script **cpio.sh -pdlm** that you can use. To use this shell script, you would reset **CPIO** as follows:

```
CPIO=cpio.sh -pdlm
```

The FILLDEF and FILLUNDEF Makefile Variables

The **FILLDEF** and **FILLUNDEF** makefile variables are used for porting the translator, like the **SZAL** makefile variable.

FILLDEF defines the predefined macro string for the target machine. **FILLUNDEF** removes the definition of the predefined macro string for the host machine.

When you build a bootstrapped translator on your host machine to use on the target machine for the final build of the C++ translator, you need to undefine the host and define the target. For example, suppose you are building a bootstrapped translator on an AT&T 3B2 for porting to an AT&T 6386 WGS. You would define the 6386 and undefine the 3B2 as follows:

```
FILLDEF=-Di386
FILLUNDEF=-Uu3b2
```

When you run **make fillscratch** on your host AT&T 3B2, you will build an intermediate translator appropriate for running on the target AT&T 6386 WGS.

NOTE FILLDEF is set to −D followed by the predefined macro string for the target machine (for an AT&T 6386 WGS, the string is "i386"). FILLUNDEF is set to −U followed by the predefined macro string for the host machine (for an AT&T 3B2 Computer, the string is "u3b2").

SZAL Makefile Variable

If you are porting the translator, you use the **SZAL** parameter to specify the location of a file containing size and alignment information for the machine on which you are installing the C++ translator. The setting of **SZAL** will be the full pathname of the output file generated in step 3 under "Porting the Translator." For example, if your output file is **/usr/patp/szal**, you would set **SZAL** as follows:

```
        SZAL=/usr/patp/szal
```

The default value of **SZAL** is null.

DENSE Variable in src/makefile

The C++ Language System encodes type information into function names, resulting in long internal names that can reach lengths of 100 characters or more in extreme cases. The Release 2.0 translator can generate names that do not exceed 31 characters, but these names will be harder for programmers to decode.

If your C compilation system does not distinguish between names of more than 31 characters, you must set the **DENSE** makefile variable to build a translator that generates shorter names. Setting the **DENSE** variable in **src/makefile** to **−DDENSE** before you run **make** will cause the makefile to build a translator that generates names of 31 characters or less.

NOTE DENSE is set to −D followed by "DENSE."

If your C compilation system does not distinguish between names of at least 31 characters, then even the DENSE option will not work. You cannot use the translator with a compilation system that does not handle at least 31-character variable names.

The following table summarizes the circumstances under which you should set DENSE:

Your C compiler distinguishes between variables of length:	set DENSE?
■ less than 31 characters	you can't use the translator
■ less than or equal to 31 characters	yes
■ greater than or equal to 31 characters but less than 100 characters	if needed
■ greater than or equal to 100 characters	no

NOTE

If your C compilation system distinguishes between names longer than 31 characters, but limits names to some other length, you may in some circumstances encounter name clashes because long names have been truncated by your compiler. In this case, you should set the DENSE variable to -DDENSE in **src/makefile** and rebuild the translator.

Building Release 2.0 on a Machine with a Translator Present

If you have an existing C++ translator on your machine, follow this procedure to build the new translator. If not, skip to "Building Release 2.0 on a Machine with No Translator."

1. Make sure the CC parameter in the makefile is set to the location of the CC command for your existing translator.
2. Decide whether you want to make a munch or a patch version of the translator. You can only make a patch version if you are running UNIX System V or BSD (the patch version is described under "The Patch and Munch Options").
3. Run *one* of the following commands:

    1. If you want to make a patch version, run the command **make patch**:

    ```
    $ make patch
    ```

    2. If you want to make a munch version, run the command **make munch**. (The **make** command with no argument builds a munch version by default.)

    ```
    $ make munch
    ```

Either procedure builds the complete C++ translator, implementing all the new features of Release 2.0. (See Chapter 4 of this document and Chapter 1 of the C++ *Language System Selected Readings* for further information on new features.)

When you start the build procedure, skip directly to the section "Expected Warning Messages During Build" to read about warning messages you may see while building. When the procedure is finished, your prompt will return. Continue with the installation procedure following "Expected Warning Messages During Build" (the next section is "Testing Results of the Build").

Building Release 2.0 on a Machine with No Translator

The following procedures explain how to build a Release 2.0 translator on a machine with no previous version of the translator. Since the C++ translator source is written in the C++ programming language, you need to create an intermediate translator to translate the source so that it can be compiled. This is called "bootstrapping" the translator.

This section describes some preliminary procedures, then tells you how to bootstrap the translator and build the new Release 2.0 translator.

Bootstrapping the Translator

The bootstrapping procedure creates an intermediate C++ translator with which to compile the C++ translator source. This procedure (called **make scratch**) uses C source files in the **scratch** subdirectories to build the intermediate translator. These source files are only appropriate for AT&T 3B computers, DEC VAX computers, or SUN 2/3/4 workstations.

If you are installing the translator on an AT&T 3B computer, DEC VAX computer, or SUN 2/3/4 workstation, you can simply go ahead and do the following steps.

CAUTION Do not continue with these instructions unless you are installing on an AT&T 3B computer, DEC VAX computer, or SUN 2/3/4 workstation. If you are not installing on one of these machines, do the steps under "Porting the C++ Translator" and regenerate scratch files (using **make fillscratch**) for your machine before continuing.

Run the program **make scratch** as follows:

```
$ make scratch
```

Running **make scratch** produces five files and a subdirectory in the current directory:

- CC, the shell script that invokes the C++ translator
- **cfront**, the executable code (main pass) for the translator itself
- **munch**, a secondary pass of the translator, which handles static constructors and destructors
- **libC.a**, an incomplete version of the C++ library
- **c++filt**, which decodes the variable name encoding in any error messages
- the subdirectory **incl**, which contains the C++ header files for your machine

make scratch also edits the existing CC command, which you use to invoke the translator.

You should now make the C++ translator from the C++ source, as described in the next sections. Note that you need to do this to complete **libC.a**.

Building the Release 2.0 Translator from C++ Source

NOTE At this point, if you want, you can run the **make clean** command to remove **.o** files created when you bootstrapped the translator. This will free space that may be needed during the build.

CAUTION If you bootstrapped the translator, building the translator as described below will overwrite the versions of **CC**, **cfront**, **munch**, and **libC.a** that you produced. We recommend, if you have space, that you save copies of these versions in case you encounter problems with building the translator.

1. Decide whether you want to make a munch or a patch version of the translator. You can only make a patch version if you are running UNIX System V or BSD (the patch version is described under "The Patch and Munch Options").
2. Run *one* of the following commands:
    1. If you want to make a patch version, run the command **make CC=`pwd`/CC patch** (setting **CC** on the **make** command line overrides any possibly incorrect setting).

       ```
       $ make CC=`pwd`/CC patch
       ```

       or

    2. If you want to make a munch version, run the command **make CC=`pwd`/CC munch.** (The **make** command with no argument builds a munch version by default.)

       ```
       $ make CC=`pwd`/CC munch
       ```

Either procedure will build a Release 2.0 C++ translator. When the procedure is finished, your prompt will return.

Expected Warning Messages During Build

When the **cfront** component of the C++ Translator is being compiled, during the generation and compilation of the **y.tab.c** file, several warning messages are generated. They need not concern the user.

```
yacc gram.y
conflicts: 7 shift/reduce, 4 reduce/reduce

CC y.tab.c
"gram.y", line 2293: warning:  statement after goto not reached
"gram.y", line 1381: warning: label yynewstate  not used
"gram.y", line 1381: warning: label yyerrlab  not used
```

Testing Results of the Build

Do the following steps to test the installation procedure:

1. If you want, run the **make demos** procedure.

   ```
   $ make CC=`pwd`/CC demos
   ```

   This procedure checks static constructors and grammar by running test programs invoking the C++ I/O library. If the programs complete successfully, then the Release 2.0 translator has been built correctly. The demonstration programs are provided mainly as examples of how to use the new C++ features and libraries.

2. If you want, run **make clean** (described under "Options to the **make** Program") to remove **.o** files generated by the installation. This procedure is entirely optional, and you may want to save the **.o** files for future use. However, if space is tight on your machine, running this procedure will free some extra space.

NOTE A good general test of the newly built C++ Translator is to have it rebuild itself. If you want to do this optional step, run **make clean** and **make install**, set the CC parameter in the makefile to point to the CC command that invokes the newly created **cfront**, and run **make** or **make patch** again.

Installing Files Created During Build

Running one of the build procedures described above produces five files and a new subdirectory in the current directory:

- CC, the shell script that invokes the C++ translator
- **cfront**, the main pass of the translator
- either **patch** or **munch**, the secondary pass of the translator
- **libC.a**, the standard C++ library
- **c++filt**, which decodes the variable name encoding in any error messages
- the subdirectory **incl**, which contains the C++ header files for your machine

You can move these files to other directories of your choice using the following procedure:

1. Make certain the INSTALL_BIN, INSTALL_LIB, and INSTALL_INC parameters in the makefile are set to the appropriate directories in which you want the translator executables to reside.

   For information on the settings of these variables, see the section "Setting Makefile Variables."
2. Run the command **make install**:

   ```
   $ make install
   ```

   This command moves the files to the directories you specified and edits the CC command so that it knows where to find the necessary executables.

The CC command should now be ready to use.

Options to the make Command

The **make** command has several options that can be run to do specific functions required during installation of the translator. These options may be useful at this point of the installation. This section describes these options and how to run them.

The make clean Procedure

The **make clean** procedure removes **.o** files generated during compilation. To run this program, enter:

```
$ make clean
```

You can use this procedure to free space taken up by these files.

The make clobber Procedure

The **make clobber** procedure removes both **.o** files and executables created during compilation. You might use this procedure if:

- something went wrong during the build and you want to remove everything and start over again, or
- you want to free space after you have run **make install**

To run **make clobber**, enter:

```
$ make clobber
```

make clobber leaves the directory structure as it was when the source was copied from the translator tape, except for the scratch files. If you have altered the scratch files, you must re-install them from the tape.

CAUTION **make clobber** removes all executables in the directory where you build your C++ translator. You should not run this option unless you want to remove all the translator executables and start over.

make clobber will not remove translator executables that have been moved to final locations using **make install**.

Building the C++ Libraries

After you have done all the installation steps described above, you can build several optional C++ function libraries. Building the translator automatically builds the new iostream library. In addition, you can build the following libraries:

- the task library
- the complex arithmetic library
- the old stream library

Setting the CC Variable

You should use your newly installed translator to build the C++ class libraries. Before building the libraries, set the **CC** parameter in your **makefile** to point to the location of your newly installed **CC** command.

Building the Task Library

The C++ task library supports co-routine programming, allowing you to run and control multiple tasks within one UNIX system process. The task library is described in detail in Chapter 2 of the C++ *Language System Library Manual.*

You can build this library by running the **make libtask.a** command. You must specify a setting for the parameter **MACH** to identify your machine when you build the task library. The settings for the **MACH** parameter are listed under "Setting Makefile Variables."

To build the task library, run the command:

```
$ make libtask.a
```

NOTE You can also set **MACH** on the command line. For example, to build the task library on an AT&T 3B Computer, run the command **make MACH=3b libtask.a**. A value set from the command line overrides any value previously set in the makefile.

NOTE If you want to build the task library on a machine other than the AT&T 3B line, the AT&T 6386 WGS, the SUN 2/3/4 workstations, or VAX computers, see Chapter 2 of the C++ *Language System Library Manual* under "Porting the Task Library."

After building the task library, you can test it by running the **make demotask** command. This command compiles five programs that use the task library and compares their output with files containing the expected output. The file **demo/task/README** lists and describes these tasking programs.

To run this command for an AT&T 3B computer, you would enter:

```
$ make demotask
```

from the top directory of the translator source.

NOTE As with **make task**, you must make sure that the **MACH** parameter is set in the makefile or from the command line.

make demotask will run **make libtask.a** if necessary, so you can build the task library and test it in one step by entering this command.

When the program finishes running, a message will confirm the success of each test and your **demo/task** directory will contain executable files corresponding to the tests described in the **README** file, with the suffix **.E** (e.g., **triv.E**).

NOTE The source for these task library tests, contained in **.C** files (e.g., **triv.C**, **shared.C**, etc.), provide examples of how to use the task library in C++ programs.

Building the Complex Arithmetic Library

The complex arithmetic library provides a data type and operators for complex numbers. This library is described in detail in Chapter 1 of the C++ *Language System Library Manual*. If you want to build this library, run the command:

```
$ make libcomplex.a
```

After building the complex aritmetic library, you can test it by running the **make democomplex** command. This command compiles programs that use the complex arithmetic library and returns messages telling you whether the programs passed or failed.

To run these demonstration programs, you enter:

```
$ make democomplex
```

from the top directory of the translator source.

make democomplex will run **make libcomplex.a** if necessary, so you can build the complex library and test it in one step by entering this command.

Building the Old Stream Library

The new iostream library, which is built automatically with the translator as part of **libC.a**, is now the default stream library. This library is described in detail in Chapter 3 of the C++ *Language System Library Manual.* The new iostream library is upwardly compatible with the old stream library, and most programs written using the old library will still run.

The old stream library is included with Release 2.0 for users who want to continue using it; however, it must be built explicitly. This library is optional for users who have existing software written under Release 1 that makes use of the internals of the library.

You can build the old stream library for use with Release 2.0 by running the command:

```
$ make libOstream.a
```

If you want to continue using the stream library delivered with Release 1.2 of the translator, you can do so by changing any **#include** statements to specify **Ostream.h**. In Release 2.0, **#include <stream.h>** and **#include <iostream.h>** will use the headers for the new iostream library.

Installing the Libraries

Once you have built the C++ class libraries, you must install them in the appropriate directory. The location of the C++ class libraries must correspond to the location specified by your newly built CC command. You should check the CC shell script, and put the libraries in the directory specified by the CCLIBDIR parameter (libC.a will already be installed in this directory).

Porting the C++ Translator

The following instructions will help you adjust your translator to make it run on some machines. In general, you must port the translator if your target machine:

- is not an AT&T 3B computer, DEC VAX computer, or SUN 2/3/4 workstation

  and

- does not have an existing translator running

To port the translator to another machine, you must have access to an AT&T 3B computer, DEC VAX computer, or SUN 2/3/4 workstation, or to an existing translator.

NOTE

The translator tape supplies header files for machines running UNIX System V Release 2 or 3, BSD 4.2 and 4.3, SUN workstations running SUNOS, and Hewlett Packard machines running HP-UX. If you are installing on a machine running another operating system, you may need to do some porting work on header files, *even if you have an existing translator on your machine.*

If you are installing on a machine with an existing translator for which the tape does not provide header files, check these instructions (particularly step 7) to determine if you need to port header files to your machine.

NOTE

Some computers with essentially the same sizes and alignments as the 3B line of computers (e.g., the DEC VAX computers) can have the translator installed with few or no porting changes.

For certain other machines, we provide **sed** scripts to alter the intermediate C language files to build correctly. For example, on BSD systems the makefile automatically invokes the script **bsd.sed** for this purpose.

You need an operating translator to port the translator to any machine for which no **sed** script is provided.

Overview

Porting requires that you have Release 2.0 of the translator running on a base machine, called the *host* machine, from which you can do the work necessary to build the translator on another machine, called the *target* machine.

Simply put, porting involves adjusting the C++ translator include files for your target machine and regenerating scratch files (the intermediate ..c files). After doing these steps, you proceed to build the translator on the target machine, as described in "Bootstrapping the Translator."

Where appropriate, this chapter tries to provide helpful hints for certain target machines based on the previous experience of users porting the translator to their machines.

You should do all of the steps in the following list to port the translator. Each step is detailed in the sections that follow this overview.

- copy the translator directory tree structure onto the host machine
- compile and run the **szal.c** program on the target machine and move a file containing its output to the host machine
- check that the size and alignment information in the **src/size.h** file is correct for your target machine

- port the C++ header files on the host machine so that they are correct for the target machine and replace the header files currently on the host machine with the ported ones
- make intermediate ..c files by running the **make fillscratch** command on the host machine
- copy the translator directory tree structure from the host machine to the target machine
- bootstrap the translator on the target machine (using **make scratch**) as described under "Bootstrapping the Translator"

Porting Instructions

Do the steps below to port your translator to the target machine. You should work on the host machine from the translator root directory.

1. Install the source for the translator on your host machine.
2. Make certain you have a Release 2.0 translator on your host machine. If you do not, you can build the Release 2.0 translator from the source by running the **make** or **make patch** procedure on the host, as described under "Building Release 2.0 on a Machine with No Translator."
3. Move a copy of the **szal.c** file (in the translator root directory) from the host machine to the target machine. Compile (with your C compiler) and run the **szal.c** file on the target machine, redirecting the output of the program to a file such as **szal**:

   ```
   $ cc szal.c
   $ a.out > szal
   ```

   When compiled on the target machine, **szal.c** generates target machine size and alignment information that can be used by the **make** program.

   Figure 3-3 shows the size and alignment information generated for an AT&T 6386 WGS Computer by compiling and running the **szal.c** program.

Figure 3-3: Output of szal.c for AT&T 6386 WGS Computer

```
$ cc szal.c
$ a.out
bit         8         32
word        4         4
char        1         1
short       2         2
int         4         4                 2147483647
long        4         4
float       4         4
double      8         4
bptr        4         4
wptr        4         4
struct      1         1
struct2     0         1
```

The first field of the **bit** line shows the number of bits in a byte. The second field shows the number of bits in a word. All other rows (except **struct2**) show the size and alignment in bytes in the first and second field, respectively. For example, in the **double** row, the first column shows a size of 8 bytes for **doubles** and the second column an alignment of 4 bytes for **doubles**.

Note that the **int** line contains a third field, which shows the maximum unsigned integer value. The first field of the **struct2** entry shows whether the bit field specification is sensitive to type (i.e., whether **int** : 2 is different from **char** : 2). The second field shows whether consecutive bit fields are packed into consecutive bits.

CAUTION **szal.c** is a C++ translator porting aid, and not part of the C++ Translator. Because it attempts to determine machine-dependent properties, unexpected dependencies or bugs on the target system may cause it to fail.

4. Move the output file you created in step 3 (i.e., **szal**) back to the host machine.

5. Edit the **makefile** file under the translator root directory on the host machine and set the following parameters:

   - set the **SZAL** parameter to the full pathname of the output file generated in step 3. For example, if your output file is **/usr/patp/szal**, you would set **SZAL** as follows:

     ```
     SZAL=/usr/patp/szal
     ```

   - set the **FILLUNDEF** parameter to the predefined macro string corresponding to your host machine. For example, if your host machine is an AT&T 3B2, set **FILLUNDEF** as follows:

```
FILLUNDEF=-Uu3b2
```

- set the **FILLDEF** parameter to the predefined macro string corresponding to your target machine (see Figure 3-2 for a list of predefined macro strings). For example, if your target machine is an AT&T 6386 WGS, set **FILLDEF** as follows:

```
FILLDEF=-Di386
```

6. Check the size and alignment information in the **src/size.h** header file under the C++ translator directory on the host machine. The size and alignment information in this file should correspond to your target machine. (The relationship between the information in **src/size.h** and the **szal** output is shown in Figure 3-6.)

   There are three ways of providing the correct size and alignment information:

   - The **src/size.h** file may already contain information specific to your target machine. In Release 2.0, it contains information for the following target machines, which are given in Figure 3-4 along with the predefined macro strings used in **src/size.h**:

Figure 3-4: Predefined Macro Strings Used in src/size.h

Machine	Predefined Macro String
AT&T 3B2 Computer	**u3b2**
AT&T 3B5 Computer	**u3b5**
AT&T 3B15 Computer	**u3b15**
AT&T 3B20 Computer	**u3b**
AT&T 6386 WGS Computer	**i386**
DEC VAX line of Computers	**vax**
IBM RT Personal Computer	**ib032**
SUN-2 Workstations	**sun2**
SUN-3 Workstations	**sun3**
SUN-4 Workstations	**sun4**
Amdahl UTS Computer	**uts**
Intel 80286 large model	**Ai286l**
Intel 80386-based computers	**i386**
Hewlett-Packard 9000 series 200	**hp9000s200**
Hewlett-Packard 9000 series 300	**hp9000s300**
Hewlett-Packard 9000 series 800	**hp9000s800**
Pyramid Computers	**pyr**
Alliant Computers	**alliant**
68000-based Apollo Workstations	**apollo**

NOTE The predefined macro strings in Figure 3-4 are only used by the translator for porting to these machines. The predefined macro string used by your native C compiler may differ.

If your machine is on this list, you probably do not need to make any changes to the **src/size.h** file, but you should check the information as described below.

- The size and alignment information for your machine may be identical to that of one of the machines listed in **src/size.h**. If the size and alignment values for the machine you are going to use correspond to a set of values already in the **src/size.h** file, you can simply add to the appropriate conditional. For example, if you were porting to a SUN-3 workstation, you would change

```
#ifdef apollo
      to
#if defined(apollo) || defined(sun)
```

NOTE The SUN 3 workstation already has an entry in the **src/size.h** file, of course, and it would not be necessary to add it; it is provided only as an example of how to add a machine that is not defined.

NOTE You will have to determine the predefined macro name for your machine (e.g., "sun" for the SUN-3 workstation in the example above). We recommend you refer to documentation for your C compiler to determine this macro string. You will need to know this macro string if your machine is not already defined in **src/size.h**, either to add to a conditional entry as above, or to add an entire size and alignment entry, as described below.

- If your machine is not one of the machines listed in **src/size.h**, and its size and alignment values do not correspond to one of these machines, you will have to add an entire **#else #if** *your_target* statement followed by an appropriate set of **#define** values before the final **#else** in **src/size.h**. These **#define** statements list the size and alignment information for your machine. Figure 3-5 shows the size and alignment information in **src/size.h** for the AT&T 6386 WGS computer.

Figure 3-5: Specifications in src/size.h File for AT&T 6386 WGS Computer

```
#if defined(vax) || defined(ibm032) || defined(i386)
                                  /* VAX, IBM 32, Intel 386 */
#define DBI_IN_WORD 32
#define DBI_IN_BYTE 8
#define DSZ_CHAR 1
#define DAL_CHAR 1
#define DSZ_SHORT 2
#define DAL_SHORT 2
#define DSZ_INT 4
#define DAL_INT 4
#define DSZ_LONG 4
#define DAL_LONG 4
#define DSZ_FLOAT 4
#define DAL_FLOAT 4
#define DSZ_DOUBLE 8
#define DAL_DOUBLE 4
#define DSZ_STRUCT 1
#define DAL_STRUCT 1
#define DSZ_WORD 4
#define DSZ_WPTR 4
#define DAL_WPTR 4
#define DSZ_BPTR 4
#define DAL_BPTR 4
#define DLARGEST_INT "2147483647"        /* 2**31 - 1 */
#define DF_SENSITIVE 0
#define DF_OPTIMIZED 1
```

Add an entry in src/size.h for your machine with the correct size and alignment values in the format shown in Figure 3-5. The easiest way to do this is to edit the file, copy the size and alignment values for another machine in the file, and then edit those values so they correspond to your machine. To find the correct values, look at the output in the file **szal**, which you created in step 3.

Figure 3-6 shows how the values in your **szal** output file correspond to the parameters you need to specify in **src/size.h**. The *italicized* labels in Figure 3-6 are parameter names from src/size.h, and appear immediately to the right of the value that corresponds to that parameter name. By taking your **szal** output and using Figure 3-6 as a template, you can match each parameter with its appropriate value for your machine.

Figure 3-6: Labels for szal Output

```
bit       8 DBI_IN_BYTE    32  DBI_IN_WORD
word      4 DSZ_WORD       4 no parameter
char      1 DSZ_CHAR       1 DAL_CHAR
short     2 DSZ_SHORT      2 DAL_SHORT
int       4 DSZ_INT        4 DAL_INT        2147483647 DLARGEST_INT
long      4 DSZ_LONG       4 DAL_LONG
float     4 DSZ_FLOAT      4 DAL_FLOAT
double    8 DSZ_DOUBLE     4 DAL_DOUBLE
bptr      4 DSZ_BPTR       4 DAL_BPTR
wptr      4 DSZ_WPTR       4 DAL_WPTR
struct    1 DSZ_STRUCT     1 DAL_STRUCT
struct2   0 DF_SENSITIVE   1 DF_OPTIMIZED
```

For example, the first column in the **long** row is the value of the parameter **DSZ_LONG**, which has the value "4." You would edit the **DSZ_LONG** line in **src/size.h** to read:

```
#define DSZ_LONG 4
```

Edit all the parameters to correspond to the parameters in **szal**, and be sure to change the **#ifdef** statement at the beginning of your entry to your machine name. For example, if you are defining the size and alignment information for an AT&T 6386 WGS computer, the first line of the definition would read **#ifdef i386.**

7. Port the header files in the **incl** directory of your host machine to the target machine and replace the header files on your host machine with the ported ones as follows:

   A. Copy the directory structure under **incl-master** from your host machine to your target machine. Change to the target machine, so that all the following steps up to step 8 are done on the target machine.

   B. Check to see if the operating system of your target machine is on the following list of settings for the **OS** parameter. This list corresponds to the settings for **OS** that are available in the top-level **makefile**.

NOTE The setting for **OS** must be one of the valid settings that corresponds to your **SYS** setting as shown in the table.

SYS Setting	OS Setting	Operating System
SYSV	svr2	UNIX System V Release 2
	svr3	UNIX System V Release 3
BSD	bsd2	Berkeley Software Distribution Release 4.2
	bsd3	Berkeley Software Distribution Release 4.3
	sun	SUN operating system
	hpux	Hewlett-Packard UX operating system

NOTE When you set OS to "sun," the **makefile** determines whether you are running SUNOS 3 or SUNOS 4 automatically. You can also explicitly set OS to "sunos3" or "sunos4" to build on a machine running SUNOS 3 or SUNOS 4, respectively.

If the target machine's operating system *is* on the list, skip the next step and proceed directly to step D.

If the target machine's operating system is *not* on the list, proceed with the next step (step C).

C. Do this step only if the operating system of your target machine is *not* in the list of settings for OS given in step B.

You will have to do a full port of the header files to new operating systems as described in this step. In order to do a full port, you need to understand how the header files for a given operating system are generated.

Suppose for a moment that the OS parameter for your target machine is listed in the top level **makefile**. To generate the header files for this machine, you would set the OS parameter in the makefile and type **make headers** (as described in step D). The top level **makefile** calls **incl-master/makefile**, passing it the OS setting. The **incl-master/makefile** checks each header file in the **proto-headers** directory to determine whether it is used by the operating system you specified. If it is, **make** transforms the header file into what it should look like on your target machine's operating system and moves the transformed header file into the directory **incl-master/incl**.

Each proto-header file in the **proto-headers** directory contains the necessary information to specify (1) which operating systems the header file is used with, and (2) how to transform the proto-header file on each of those operating systems. For example, consider the following proto-header file **foo.h** (line numbers are for reference):

```
1       #usedby sun hpux svr3
2       #ifndef FOOH
3       #define FOOH
4       #hide bar
5       #hide baz
6       #expand foo.h
7       extern "C" {
8               char *bar(int);
9       #os sun hpux
10              char *baz(int);
11      #endos
12      #os svr3
13              char *baz(long);
14      #endos
15      }
16      #endif
```

The following summary explains the function of the significant lines in **foo.h**:

line 1.	The **#usedby** directive specifies which operating systems a header file is used with. The header file **foo.h** is used with sun, hpux, and svr3. The **#usedby** directive should appear once, somewhere near the top of the proto-header.
lines 2, 3, 16	These lines are the standard header file "wrapper."
lines 4, 5	The #**hide** directive is used to hide C-style prototypes that appear in the target machine's native C header files. Since C++ has strong type checking, these C-style prototypes are usually wrong. There should be a separate #**hide** directive for each C-style prototype in the native C header file.
line 6	The #**expand** directive specifies the place in the header file where the native C header file on the target machine should be physically inserted. Each proto-header should contain a single #**expand** directive with the pathname of the proto-header as its argument (the path must locate the proto-header file relative to the directory **/usr/include**). The #**expand** directive should appear after all #**hide** directives, but before the **extern** "C" block.
lines 7-15	The **extern** "C" block specifies the C++-style prototypes for all functions relevant to this header. Since different operating systems can have different prototypes, #**os** directives specify which prototypes go with which operating systems. In foo.h, the prototype **char *baz(int);** is correct for **sun** and **hpux**, while **char *baz(long);** is correct for **svr3**. (**char *bar(int);** is correct for all three.) #**os** directives can appear anywhere in the proto-header, but they cannot be nested.

To port the header files to a new operating system, you must do the following:

- For each header file used with the new operating system, add the operating system's name to the #**usedby** directive of the proto-header, and fix the prototypes appropriately, using #**os** directives to avoid introducing errors into the information for other operating systems.

- Add the new operating system to the set of porting targets in **incl-master/makefile**. If you edit **incl-master/makefile** you will see a list of the machines for which **OS** can be set (as described in step B). Add your porting target to this list. For example, suppose you are adding an Amdahl UTS to the list of targets. The excerpt below shows an entry for the Amdahl added to the other entries:

```
sun:            always
                @build sun ${STDINCL} ${SYMBOLICLINK} ${VERBOSE} ${TARGET}

# new target for Amdahl UTS:
uts:            always
                @build uts ${STDINCL} ${SYMBOLICLINK} ${VERBOSE} ${TARGET}
```

- If you need to do any operating system dependent adjustments, insert them in the **build** shell script, in the section entitled "Operating System Dependent Adjustments." (The **incl-master/build** shell script is called by the makefile to drive the entire build process; the actual transformation of proto-header to C++ header is done by the shell script **transform** and the executable file **expand**.)

Now that you have finished the complete port of the header files for your target machine, continue with the build procedure described in step D.

D. Set the following parameters in **incl-master/makefile** to their appropriate settings:

- set the value of the **OS** parameter to the appropriate setting for the operating system of your target machine given in the table in step B (if you are porting to a machine not in this table, the setting should be the predefined macro string for the new machine as described in step C)
- set the value of the **SYMBOLICLINK** parameter to "0"

E. Run the command **make** from the **incl-master** directory on the target machine:

```
$ make
```

Running **make** populates the directory **incl-master/incl** with C++ header files for the target machine.

F. Move a copy of the directory structure rooted at **incl-master** from the target machine to the **incl-master** directory of the host machine.

G. Edit the top level makefile, changing the **OS** entry to correspond to your target machine (you might have to add a new **OS** entry). For example, if your target machine is an Amdahl UTS computer, and the predefined macro string for the machine is **uts**, you would add the entry as in the following excerpt:

```
#OS=sun
OS=uts
#OS=hpux
```

H. NOTE Before running **make incl**, which will copy the directory structure rooted at **incl-master** to the top level **incl** directory, you may want to save the current contents of **incl** on the host by copying them to another directory.

From the top level directory of your host machine, run the command **make incl**:

```
$ make incl
```

This command moves the files in **incl-master/incl** to the top level **incl** directory on your host machine.

8. On your host machine, make certain the parameter **CC** in the top level makefile points to the existing C++ translator on the host machine.
9. On your host machine, set and **export** the parameter **I** in the **CC** shell script to point to the incl directory under your working directory.
10. Execute **make fillscratch** from the translator root directory on your host machine.

```
$ make fillscratch
```

Running this command repopulates the **scratch** directory with the intermediate **..c** files.

11. Copy the translator source (i.e., everything under your translator root directory) from your host machine to your target machine.

NOTE The entire directory structure of the translator is about 6000 blocks. In moving files from one machine to another, you may encounter problems with your **ulimit**. If this happens, moving the translator source in smaller pieces should resolve the problem.

12. Make sure the parameters in your makefile are set correctly for your *target* machine, as described under "Setting Makefile Variables."
13. Check the sections "Miscellaneous Adjustments" and "Common Problems" to see if there are any other adjustments you should make for your target machine before you build the translator.

14. Build the translator on the target machine from the ..c files as detailed under "Bootstrapping the Translator," (which describes how to run **make scratch**), and continue with the installation procedure from that point.

Miscellaneous Adjustments

The following sections describe adjustments that may be required before bootstrapping the translator.

UNIX System Shells

The CC command is a shell script written under the UNIX System V Bourne Shell. It may not work correctly under every UNIX Shell variant. The **#!/bin/sh** first line in **CC** causes the BSD system to invoke **/bin/sh** rather than, say, **/bin/csh**.

The Amdahl UTS may require a colon (:) as the first character of the shell script file to invoke **/bin/sh**. If this is a problem on your system, you will have to edit the **CC** command file.

Non-UNIX Systems

The C++ translator has been successfully ported to machines supporting operating systems other than the UNIX System. However, since the **CC** command is made up of UNIX system shell commands, **CC** will probably require significant changes if it is to run under a non-UNIX system.

In addition, the **CC** command uses the UNIX system command **nm** to pass the symbol table of the just-compiled program to **munch**. **munch** inspects the symbol table to see if the program has any static constructors or destructors. Any port of the C++ translator to a non-UNIX system will have to replace the call to **nm** in **CC** with a functionally similar command. See the **munch** source in **lib/static/munch.c** for further details.

Environment Information

Release Date Stamp

The official release date of each release of the C++ Language System is encoded in the **cfront** command. The date stamp of each release is as follows:

- Release 1.0: 10/10/85
- Release 1.1: 5/20/86
- Release 1.2: 1/15/87
- Release 2.0: 6/30/89

To see the date stamp for your system, you may either invoke **cfront** directly, or look in the intermediate ..c C language file the C++ Translator generates when you run the command CC **-Fc** -..c *file*.c.

Setting Environment Variables for the CC Command

The CC script uses environment variables to locate files it needs to run. You can override the values of these environment variables by setting them to different locations and **exporting** them. For instance:

```
$ ccC=/usr/my/cc;export ccC
```

will use /usr/my/cc as the C compiler instead of the default cc.

The following table lists the most important environment variables used by CC:

Environment Variable	Default	Function
ccC	cc	C compiler
cppC	/lib/cpp	C preprocessor
I	/usr/include/CC/	directory for include files
CCROOTDIR	/usr/bin	directory containing **cfront, c++filt, patch** or **munch**, and the CC command
CCLIBDIR	/usr/lib	directory containing the C++ libraries

Common Problems

This section lists problems that can occur when trying to build the C++ translator.

We recommend that you look through this list before beginning to build or port your translator. Checking this list will help you avert potential problems or recognize the source of any problems that do occur.

Wherever a know workaround exists for the problem described, it will be explained.

General Build Problems

The following problems may occur when trying to build the translator on any machine.

- **Problem:** Error message "free store exhausted" occurs.

  Solution: Your machine does not have enough memory to build the translator. The C++ translator can grow to over 500 kilobytes when it is compiling itself. If this is too large for your **ulimit(2)**, have a system administrator raise the limit for your processes.

- **Problem:** Error message about a **yacc** overflow occurs.

  Solution: The C++ grammar may overflow the internal tables on some **yacc** files. You will need to ask your system administrator to rebuild **yacc** with larger tables.

- **Problem:** Compiler can't distinguish between variable names. This will show up as an error message that two names which differ only in the last character positions are redeclared.

  Solution: The C compiler on your system must be able to distinguish variable names of at least 31 characters to use the translator. If your compiler distinguishes between less than 100 but more than 31 characters, you must set the **DENSE** makefile variable to **-D** followed by "DENSE" in your makefile before building the translator.

  If your compiler cannot distinguish at lease to 31 character positions, then you cannot use the translator.

- **Problem:** An internal compiler error "tree size exceeded" occurs.

  Solution: The default maximum tree size of your C compiler is not large enough. Ask your system administrator to increase the tree size variable in your C source and rebuild the C compiler.

Machine-Specific Build Problems

The following problems may occur while installing the translator on a particular machine.

AT&T 3B Line of Computers

- **Problem:** Any attempt to use the task library dumps core. This happens on all UNIX System V Release 2 operating systems running on WE32000-based machines (e.g., AT&T 3B15 computers). This problem does not occur under System V Release 3.

  The problem occurs because System V Release 2 on WE32000-based machines did not allow the stack pointer to point into the free store. Because DEDICATED tasks rely on this capability, they cannot be used on these systems.

Solution: Define the _SHARED_ONLY parameter before building the task library from source. To do this, set the **CCFLAGS** parameter in the top level **makefile** as follows:

```
CCFLAGS=-D_SHARED_ONLY
```

This will build a version of the task library that only uses SHARED tasks.

AT&T 3B2 Computers

- **Problem:** The error message "mau hardware required" occurs when trying to use executables produced by C++.

  Solution: 3B2 binaries are built for systems with a MAU (math accelerator unit). If your machine does not have a MAU, rebuild the translator, setting **CCFLAGS="$CCFLAGS -f"** to invoke the floating point emulator. Otherwise, contact support to obtain a MAU that does floating point emulation.

386-Based Computers

- **Problem:** Programs using the task library fail when a signal handler must be called. This happens on all Intel 386-based machines running UNIX System V Release 3 with DEDICATED tasks. (System V Release 3 on 386-based machines does not call signal handlers when the task is running on a stack in the free store, i.e., a DEDICATED task.)

  Solution: Define the _SHARED_ONLY parameter before building the task library from source. To do this, set the **CCFLAGS** parameter in the top level **makefile** as follows:

  ```
  CCFLAGS=-D_SHARED_ONLY
  ```

  This will build a version of the task library that only uses SHARED tasks.

Amdahl UTS Computers

- **Problem:** When trying to use C++, a fatal error in /lib/c2 occurs or internal compiler errors occur.

  Solution: This problem has occurred when trying to compile C++ programs with the optimizer (-O) flag. The compiler has bugs in it when it tries to optimize C++ code. You should edit the top-level makefile and remove the **-O** value from the **CCFLAGS** variable.

Hewlett-Packard 9000 Computers

- **Problem:** The C compiler will give an internal error when building **dcl4.c** and possible other routines.

  Solution: There is a bug in the C compiler on Hewlett-Packard machines when the **-O** (optimizer) option is used. Do not use the optimizer when building. To remove the optimizer feature, edit the makefiles **src/makefile** and **lib/mk/makefile** and remove the **-O** value for the **CCFLAGS** parameter.

4 Compatibility with Previous Releases

Compatibility with Previous Releases

Release 2.0 of the C++ Language System is generally source compatible with C++ programs running under previous releases of the C++ Language System.

This chapter lists new features of Release 2.0 of the C++ Language System that may require changes to existing C++ programs running under Release 1.2. In general, the new features added since Release 1.2 do not require the rewriting of existing code. These upwardly compatible new features are not described here, but are discussed in detail in Chapter 1 of the C++ *Language System Selected Readings*. This chapter focuses on changes since Release 1.2 that may require source code changes.

However, all existing code will need to be recompiled using the Release 2.0 Language System to link with new code compiled under Release 2.0.

This chapter also describes changes in procedures for building, installing, and using the C++ Language System.

This chapter is organized into the following sections:

- *Building the Translator* — tells you information you must know before installing the C++ translator
- *Header Files* — tells you about changes to C++ Language System header files in Release 2.0
- *Changes to the* CC *Command* — tells you about new options to the CC command, macro name changes, and other changes in functionality
- *Language Extensions and Changes* — tells you about changes and extensions to the language that may affect the usability of C++ programs written under previous releases of the Language System. (Other language extensions that do not affect the compatibility of code written under previous releases are described in Chapter 1 of the C++ *Language System Selected Readings*.)
- *Library Changes* — discusses new libraries supplied with Release 2.0 and changes in their organization

Recompilation of Old Code Required

Previously compiled code should be recompiled using Release 2.0. The internal name encoding algorithm that the translator uses in Release 2.0 is considerably different from the encoding algorithm used in previous releases. Additionally, the implementation of multiple inheritance necessitated some changes in object layout. Therefore object files and libraries compiled with previous releases of the Language System will not be compatible with object files compiled under Release 2.0.

Building the Translator

NOTE The procedures for building and installing the Release 2.0 translator have been streamlined and simplified. The **makefile** will do more things automatically, but first you must edit the top level **makefile** and change the settings of the user-tunable variables as needed for your environment. Please read Chapter 3 for details.

The following tips should be noted before building the Release 2.0 translator.

The FDOTRIGHT Macro

Some C compilers prohibit expressions like **f().x**, where **f()** is a function that returns a **struct** with a member **x**. This is valid ANSI C, but the original Kernighan and Ritchie *C Reference Manual* didn't say anything about it, so some compiler writers didn't implement it. To cope with such compilers, the translator no longer assumes that the C compiler will get **f().x** right. Whenever it needs to generate C that looks like **f().x**, it generates **(temp=f(),&temp)->x** instead.

If your C compiler allows expressions like **f().x**, add **−DFDOTRIGHT** to the list of **CCFLAGS** in the **makefile** as follows:

```
CCFLAGS = -O -D$(SYS) -DFDOTRIGHT
```

This builds a translator that assumes it can produce such expressions and will therefore omit these temporaries (and associated overhead).

DENSE Makefile Variable

The C++ Language System encodes type information into function names, resulting in long internal names that can reach lengths of 100 characters or more in extreme cases. The Release 2.0 translator can generate names that do not exceed 31 characters, but these names will be harder for programmers to decode. If your C compilation system does not distinguish between names of more than 31 characters, you must set the DENSE makefile variable to build a translator that generates shorter names. Setting the DENSE variable in **src/makefile** as follows:

```
DENSE=-DDENSE
```

will cause the **src/makefile** to build a translator that generates names that do not exceed 31 characters.

If your C compilation system does not distinguish between names of at least 31 characters, then even the **DENSE** option will not work. You cannot use the translator with a compilation system that does not handle at lease 31-character variable names.

NOTE If your C compilation system distinguishes between names longer than 31 characters, but limits names to some other length, you may in some circumstances encounter name clashes because long names have been truncated by your compiler. In this case, you should set the DENSE variable in **src/makefile** to DENSE=-DDENSE, as above, and rebuild the translator.

RETBUG Parameter No Longer Needed

The **RETBUG** parameter, which was formerly used as a workaround for a C compiler bug, is no longer needed and has been removed from **src/makefile**.

Header Files

C++ programs can use C functions supplied by the underlying C compilation system and the operating system, but the header files defining the interface usually will need to be modified to be usable by C++ programs. In particular, C++ requires that all functions be declared using function prototypes and that all C functions be declared to have C linkage by using **extern** "C" declarations. (See the section on type-safe linkage, below, and Chapter 6 of the *C++ Language System Selected Readings*, for more information on **extern** "C" declarations.)

Header Files Have Been Restructured

In previous releases of the C++ Language System, C++ versions of C header files were supplied in the incl directory. These headers were based on the standard C headers supplied for AT&T C compilers on UNIX System V, and required porting for different C compilation systems and operating systems. In Release 2.0, C++ versions of C header files are automatically generated by the **makefile** using the standard C header files on your machine along with prototype header files supplied in the **incl-master/proto-headers** directory. These prototype header files identify which operating systems supply which headers, and also provide the function prototype declarations required for use with C++. Release 2.0 supplies prototype header files for these operating systems: UNIX System V Releases 2 and 3, SUN OS, and Hewlett-Packard UX Operating System.

NOTE If you use a different operating system, you will have to port your system's standard header files for use with C++. See Chapter 3, under "Porting the C++ Translator" for instructions.

The **makefile** now uses the **OS** variable to specify which header files should be generated:

OS Setting	Operating System
bsd2	Berkeley Software Distribution Release 4.2
bsd3	Berkeley Software Distribution Release 4.3
svr2	UNIX System V Release 2
svr3	UNIX System V Release 3
sun	SUN operating system
hpux	Hewlett-Packard UX operating system

NOTE The standard header files on most machines are supplied by both the C compilation system and the operating system. Using the operating system as the determinant for which header files to generate is correct in most cases, but in some cases you may need to port header files based on the C compilation system on your machine.

stdio.h Contains Only I/O Function Declarations

In Release 1.2, several commonly used C function declarations, which were not in any standard C header file, were placed in **stdio.h** for C++. In addition, some of these declarations were repeated in **libc.h**, which contained declarations for functions in the standard C library that were not declared in any standard C header file.

ANSI C now specifies that standard C library functions be declared in **stdlib.h**. In Release 2.0, function declarations have migrated to their correct header files. In particular, **stdio.h** contains only I/O function declarations. **stdlib.h** contains additional standard C library function declarations, as specified by the ANSI C standard. **libc.h** is retained for compatibility with previous releases and contains the same function declarations as **stdlib.h**, except that **stdlib.h** contains declarations for the **malloc(3)** family of functions while **libc.h** does not.

The function declarations that have been removed from **stdio.h** are: **exit()**, **abort()**, **atoi()**, **atof()**, and **atol()**. These declarations are all in **stdlib.h** and **libc.h**. New code calling any of these functions should include **stdlib.h**. Old code that included **stdio.h** to obtain any of these function declarations must be changed to include **stdlib.h** as well.

NOTE The header file restructuring described above changed the locations of other function declarations for certain operating systems. If you get an error message about an undeclared function, **grep** the header file directory to find which header file contains the declaration you need.

Changes to the CC Command

The following changes have been made to the **CC** command and how it compiles C++ source files.

+a Option

The translator can now generate either ANSI C style or "Classic C" (also known as K&R C) style declarations. A new option, **+a**, has been added to the **CC** command to specify which style of declaration should be produced. This is primarily to allow the use of single precision floating point computation on systems that have an ANSI C compiler that supports it. In addition, initialization of automatic aggregates (aggregates at local scope) is only implemented under the **+a1** option, as only ANSI C compilers handle the generated C code. **+a0**, the default, causes the translator to generate "Classic C" style declarations, while the **+a1** option causes it to generate ANSI C style declarations.

+p Option

The CC command now accepts a **+p** option (for "pure") under which the translator disallows any anachronistic constructs, such as assignment to **this**. See the *AT&T C++ Language System Product Reference Manual*, Appendix B under "Anachronisms," and Appendix C, "Implementation Specific Behavior," for lists of anachronisms.

+w Option

The CC command now has a **+w** option that makes the translator produce warnings about constructs that are occasionally (but not always) questionable. Without the **+w** option, the translator only issues a warning if it is almost certain that an error exists.

+S and +V Options

The CC command options **+V** and **+S** are no longer supported. These commands caused the translator to accept C function declarations and print information to **stderr**, respectively.

CC Command Uses /usr/tmp

/usr/tmp

The CC command no longer by default generates a **..c** file in the user's working directory. It now generates a **.c** intermediary C language file in the **/usr/tmp** directory.

This change permits object language tools such as profilers and debuggers to find the C source language file under the expected **.c** name. The **+i** or **-Fc -..c** options will still, however, generate the intermediate C file with the ..c suffix in the user's directory.

__cplusplus Macro

For ANSI C compatibility, the macro definition **c_plusplus** has been changed to **__cplusplus**. For backward compatibility, **c_plusplus** has been retained for (at least) this release. New code should use **__cplusplus**.

The **__cplusplus** macro is defined in the **CC** command and allows the mixing of "Classic C" and C++ in the same header.

```
#ifdef __cplusplus
            int printf(char*...);           /* C++ function declaration */
#else
            int printf();                   /* C function declaration */
#endif
```

Language Extensions and Changes

Release 2.0 implements a number of C++ language extensions and changes. Many of the extensions are upwardly compatible with C++ Release 1.2, but some may require changes to code written under Release 1.2. In addition, bug fixes to the translator may break some code which used to be accepted, but should never have been accepted. The following sections describe changes in Release 2.0 that may affect code written under previous releases of the C++ Language System. A comprehensive discussion of changes in the language is provided in Chapter 1 of the C++ *Language System Selected Readings*.

Implicit Overloading and Type-Safe Linkage

In Release 2.0, two or more functions with the same name are overloaded automatically, and overloading is now independent of the order in which the functions are declared. In addition, the default function linkage is C++ linkage, rather than C linkage; C functions must be declared explicitly to have C linkage. Implicit overloading is implemented by encoding all C++ functions with type information, which in turn allows function calls to be type-checked across files. These changes may necessitate a few changes in existing programs.

In previous releases, a function had to be explicitly overloaded. The first instance of an overloaded function was given C linkage and subsequent instances given C++ linkage. This enabled the user to overload library function names and still link with an existing C library by declaring its instance first. For example:

```
overload abs;                  // Release 1.2 syntax
int abs( int i );
double abs( double d );
complex abs( complex c );

main() {
        int i = abs( i );
}
```

Declaring **int abs(int)** first meant that it was given C linkage. This order dependency was problematic, especially when header files were mixed. Instead, in Release 2.0, you must explicitly give a function C linkage by declaring it **extern "C"**:

```
double abs( double );
extern "C" {                   // Release 2.0 syntax
        int abs( int );
}
complex abs( complex );

main() {
        int i = abs( i );
}
```

In the new syntax, the function declared within the **extern "C"** block (here, the function **abs** taking an int argument) is given C linkage, and will link with the **abs()** function defined in the C library. The **overload** keyword is no longer necessary as all functions with C++ linkage are implicitly overloaded. The **overload** keyword is now ignored.

NOTE The **overload** keyword is now an anachronism. It is still accepted, but may not be accepted in future releases.

Any C library functions must now be declared using the **extern** "C" syntax.

NOTE It is best to **#include** standard header files to obtain declarations of C functions. This ensures that the declaration uses the correct prototype and avoids errors resulting from conflicting or inaccurate declarations.

A given function name may only be declared once with C linkage. Any other functions by the same name (e.g., **abs()** declared for **double** and **complex**) must be declared outside the **extern** "C" block, and will be overloaded automatically by the translator.

Note that all C++ header files supplied with Release 2.0 contain the necessary **extern** "C" declarations. If you use any user-created header files from previous releases, be sure to convert any declarations of C function names to the new **extern** "C" syntax.

For a complete discussion of type-safe linkage and overloading, see Chapter 6 of the C++ *Language System Selected Readings*.

NOTE You cannot use **extern** "C" at a local scope.

All Functions Must Be Declared before Use

Previously, a function that was used but not declared caused the translator to produce a warning message. In Release 2.0, this condition produces an error message, and code using undeclared functions will not compile. Often this problem can be corrected by **#includeing** the header file that contains the needed function declaration.

Overload Function Matching Improved

The mechanism for overloaded function matching (deciding which of a set of overloaded functions to call) has been revised in Release 2.0. The new scheme is more expressive and catches more ambiguity errors. It allows programmers to overload functions based on arguments that used to be considered "too similar" to distinguish, such as **float** and **double**. For example:

```
int f(double);
int f(float);
```

The second declaration of **f()** would produce a warning under Release 1.2 that the overloading mechanism cannot tell an **int (double)** from an **int (float)**. It would then use a heuristic that chose the first match (**f(double)**) to resolve any ambiguous call to **f()**.

Release 2.0 can distinguish between **floats** and **doubles** and other similar types in function matching. Furthermore, all function matching is independent of the order in which functions are declared.

Additionally, the function matching semantics for overloaded and non-overloaded functions are identical under Release 2.0 (the rules for overloaded functions used to be stricter than those for non-overloaded functions). In Release 1.2, a standard conversion was not applied to an argument of an overloaded function if it involved a truncation. For example:

```
overload f;
void f (char *);
void f (int);

// In Release 1.2:  error:  bad argument list
// In Release 2.0:  warning:  float passed as int
f(3.14159);
```

The call to **f()** with a **float** argument is disallowed by Release 1.2, because the standard conversion from a **float** to an **int** involved a truncation. This was incompatible with C semantics, and with the semantics of non-overloaded functions. In Release 2.0, these incompatibilities have been resolved. The translator warns about the truncation, but performs the standard conversion on the argument to **f()**.

Because Release 2.0 counts all matches that require a standard conversion on an argument to be equally good matches, some calls that used to be accepted may be rejected by Release 2.0 as ambiguous. Programmers can resolve such ambiguities by using explicit casts or by adding additional function declarations.

See the section on "Overloading Resolution" in Chapter 1 of the C++ *Language System Language Manual* for more complete details on function matching.

() No Longer Matches (...) in Function Declarations

In Release 2.0, a function declaration followed by empty parentheses (e.g., **f()**) no longer matches a function declaration followed by parentheses containing an ellipsis (e.g., **f(...)**). A function declaration with no argument prototype declares a function that takes no arguments.

New Keywords

Release 2.0 reserves two new keywords: **template** and **catch**. These are reserved in anticipation of future implementations of parameterized types and exception handling, respectively. Any code using these words as identifiers will have to be changed. Use of these words will usually provoke a "not implemented" error message (depending on the context).

Default Initialization and Assignment Is Memberwise Copy

Beginning with Release 2.0, default assignment and initialization is memberwise copy, rather than bitwise copy. This means that if no assignment operator or copy constructor is defined for a class, and one is needed, the translator will generate the needed operator or constructor. Each member of the class will be assigned or initialized using either a user-supplied or translator-generated assignment operator or copy constructor. These rules are applied recursively to members.

NOTE

A copy constructor for a class **X** is a constructor that can be called with a single argument of type **X**, for example **X(X&)**.

operator=() Must Be a Member Function

The Release 2.0 translator will generate a warning for any definition of **operator=()** that is not a member function.

NOTE

Global **operator=()** is an anachronism and may not be accepted in future releases.

Assignment to this Is an Anachronism

In Release 2.0, the mechanisms for user-defined memory management have been refined and extended, eliminating the need for "assignment to this." In particular, the **new** and **delete** operators can be defined as class member functions, **operator new()** can be overloaded to take additional arguments, **operator delete()** can be defined to take a size argument that will be supplied automatically by the translator, and destructors can be called explicitly. These facilities are described in detail in Chapter 1 of the *C++ Language System Selected Readings.*

NOTE

Code with constructors that assign to **this** will be accepted with a warning in Release 2.0, but may not compile in future releases.

In addition, it is illegal in Release 2.0 to take the address of **this.**

Return Types for User-Defined Conversions and Destructors Now Illegal

Return types for user-defined type conversions and destructors are no longer allowed in Release 2.0. For example, the following syntax was allowed by Release 1.2:

```
class String {
        String*  operator String*();    // Release 2.0 syntax error
        void     ~String();             // Release 2.0 syntax error
} ;
```

but it is illegal in Release 2.0. You can fix this syntax error by removing the return types from the declaration:

```
class String {
        operator String*();
        ~String();
} ;
```

Visibility of Operator Functions Now Strictly Enforced

In Release 2.0 the protection level of user-defined operators is now strictly enforced by the translator. Previous releases of the translator did not check the protection level of user-defined operators. For example, the following illegal program compiled under Release 1.2, but will fail under Release 2.0:

```
class X {
        operator int ();
};

int f(X& x)     { return x; }     // used to work, but shouldn't
```

The operator definition in this class definition defaults to private because no **public** keyword is provided. Therefore, since the function **f()** is not a member function of **X**, it cannot use the private operator **int** to convert **x** to an **int**. Adding an explicit **public** keyword to the class definition:

```
class X {
        public:
        operator int ();
};
```

allows access to the operator and resolves the problem.

const Checking and const Member Functions

Before Release 2.0 **const** checking was incomplete. It was possible to declare a **const** class object and then call a member function for that **const** object to change its value. For example:

```
class X {
        int data;
public:
                X()             { data = 0; }
        int     get_data()      { return data; }
        void    set_data(int i) { data = i; }
};
void f()
{
        X xobj;
        const X const_xobj;

        int i = xobj.get_data();
        i = const_xobj.get_data(); // Read-only access
        xobj.set_data(10);
        const_xobj.set_data(10);   // Attempt to modify const object!
}
```

In previous releases, **set_data()** could be called to modify the constant object **const_xobj**. This problem has been resolved in Release 2.0 by the addition of **const member functions**. In Release 2.0, the above code produces a warning:

```
"prog.c", line 14: warning: non const member function X::get_data() called for const object
"prog.c", line 16: warning: non const member function X::set_data() called for const object
```

NOTE In future releases this warning will become an error.

In Release 2.0, member functions can be declared **const**, which means that they can be called for a const object, such as **const_xobj** in the example above. The correct way to declare **X::get_data()** in the example above would be as follows:

```
int get_data() const { return data; }
```

This declaration will eliminate the first warning. The translator will check whether a **const** member function modifies the object, and will produce an error if it does. So, for example, if **X::set_data(int)** were declared **const**, Release 2.0 would produce the following error message:

```
error: assignment to  member X::data of const struct X
```

Constructors Are Not Inherited

In previous releases, the following erroneous code was accepted:

```
struct B {
        B(int);
};

struct D: B { };

void f()
{
        D d(3); // error: no D::D(int) has been defined.
}
```

This is clearly an error, since constructors are not inherited, and the Release 2.0 translator will no longer accept this code. This may cause some programs (which should never have worked) to break. Explicitly defining a **D::D(int)** constructor will fix this error.

Default Argument Initializers Can Be Specified Only Once

In Release 2.0, default argument initializers can be specified only once in a file. For example, the following code was accepted in Release 1.2:

```
extern int f( int i = 0 );

int f( int i = 0 )              // error in Release 2.0
{ return i * 2; }
```

but now produces an error:

```
line 3: error: two initializers for f() argument i
```

We recommend putting default argument initializers in the header file, as that is generally where users will see it.

Implicit Initialization of Static Data Members is an Anachronism

In previous releases of the C++ Translator, while it was possible to declare static data members of a class, it was not possible to initialize them, nor were they allowed to be of a type with a constructor. Static data members were implicitly defined and initialized to 0.

In Release 2.0, static data members must be initialized explicitly, and there is a new syntax for doing so. In addition, static data members may be of a type with a constructor.

```
//File X.h
class X {
public:
        static int          gdata;
        static X gX;
        X()     { gdata++; }

};

//File X.c
int X::gdata = 0;               // initialize X::gdata
X X::gX;         // calls X::X constructor to initialize X::gX
```

Like other global data, static data members of classes must be initialized exactly once. Therefore, such initializations should generally not be in header files.

For compatibility with previous releases, uninitialized static data members are implicitly initialized to 0 in Release 2.0, unless the +p option is used.

NOTE Implicit initialization of static data members to 0 is an anachronism. Code failing to initialize static data members may not compile under a later release.

Because failure to initialize a static data member is not detected until link time, the translator issues no warnings about such failures.

Linkage Changes

Default linkage for all **const** objects is static, and this is enforced in Release 2.0. That is, **const** objects that are meant to be used in other files must be explicitly declared **extern**. For example:

```
extern const int global_var;
const int global_var = 10;
```

Global anonymous unions (unions declared at file scope) must be declared static in Release 2.0. For example, the following code was accepted under Release 1.2, but will produce an error under Release 2.0:

```
union    { int i; double d; }  // error in Release 2.0
```

This error can be corrected by inserting the keyword **static** at the beginning of the line.

Member functions are implicitly **extern** and can no longer be defined as having static linkage.

NOTE Don't confuse member functions having static linkage with static member functions. Static member functions are a new feature in Release 2.0. See Chapter 1 of the C++ *Language System Selected Readings* for further details.

For example, the following code was legal in Release 1.2, but is an error in Release 2.0:

```
class X {
public:
        void f();
};
static  void X::f() { }       // error in Release 2.0
```

Additionally, functions declared to be **friends** of a class implicitly have external linkage in Release 2.0. These can, however, be explicitly declared to have **static** linkage before the **friend** declaration. For example:

```
static void sf();

class X {
friend void sf();       // sf remains static
friend void f();        // f implicitly declared extern
};
static  void sf() { }   // OK
static  void f() { }    // error:  f declared as both static and extern
```

Similarly, it is illegal in Release 2.0 to declare a function or object as **extern** and then to redeclare it as **static**. The reverse, however, generates only a warning. For example:

```
extern void f();
static void f();        // error in Release 2.0

static void g();
extern void g();        // warning in Release 2.0
```

Release 1.2 accepted both constructs without warnings or errors.

Changes to Enumerations

Enumerations are distinct types in Release 2.0. This means that you can overload a function when the arguments differ only in the **enum** argument they take. For example:

```
enum e1 { a, b, c };
enum e2 { d, e };
void f(e1);
void f(e2);

void g(int);
void g(e1);
```

In addition, because they are distinct types, assigning an **int** to an **enum** will produce a warning in Release 2.0.

NOTE Assignment of an **int** to an **enum** is accepted with a warning in Release 2.0 for compatibility with previous releases, but it will become an error in future releases.

However, an **enum** can be assigned to an **int**, because it is a standard integral promotion.

```
enum small_ints { one=1; two, three, four, five };
const small_ints half_doz = 6;      // warning:  type mismatch
const small_ints smallest = one;
int i = smallest;                   // okay, standard integral promotion
```

An enumerator is entered in the scope in which the enumeration is defined. For example, if an enumerator is defined in a class, that enumerator is visible in that class's scope, subject to the usual access rules. For example:

```
class X {
        enum { x, y, z };
public:
        enum { a, b, c };
        // ...
};
void f()
{
        int i = a;              // error:  a is undefined; X::a is not in scope
        i = X::a;               // okay
        i = X::x;               // error:  X::x is private
}
```

An enumerator cannot be declared as an enumerator for more than one enumeration. For example, the following code, which was accepted with a warning in Release 1.2, will produce an error in Release 2.0:

```
enum small_ints { one=1, two, three, four, five };
enum low_numbs { zero, one }; // error:  one declared in two enumerations
```

Finally, forward declarations of **enum** specifiers were accepted in Release 1.2, but are errors in Release 2.0. For example:

```
enum E;                     // error:  forward declaration of enum E
enum E { a, b, c };
```

Cannot Adjust Access Protections of Base Members

In Release 1.2, it was possible to adjust the access protections of base members in a derived class. In Release 2.0, it is illegal to grant access to base members that was not already granted by the base class, and it is also illegal to remove access that was granted by the base class. For example:

```
class B {
        int priv;
public:
        int pub1;
        int pub2;
};

class D : public B {
        B::pub1;          // Release 2.0 error:  B::pub1 is public
public:
        B::priv;          // Release 2.0 error:  B::priv is private
};
```

However, it remains legal in Release 2.0 to restore access in the derived class to what it was in the base class. This is useful in a private derivation of a base class. For example:

```
class B {
        int priv;
public:
        int publ;
        int pub2;
};

class D2 : private B {
public:
        B::publ;         // OK, was public in base
};
```

In this case, the private derivation of **B** in **D** makes the access of all **B's** members private. Public access to **B::publ** can be restored with an explicit declaration.

Allocation Failures

It is guaranteed in Release 2.0 that if **new** runs out of memory, it will not call the constructor. Constructors no longer need to test for **this!=0** to forestall core dumps.

Size Argument to Overloaded new and delete

If you define your own operator **new** or **delete**, the size argument must be of type **size_t**, not **long.**

The ANSI C standard defines **size_t** as an unsigned integer big enough to hold an address. **size_t** is the type returned by **sizeof()** and also the type of the argument to **malloc()**. It is defined in **stddef.h** and also (for C++) in **new.h**. For most machines, it will be **unsigned int**, but many MS-DOS implementations will use **unsigned long** for the big memory models.

If you define an operator **new** or **delete** with a **long** argument, you get a warning message and the translator will convert it to **size_t** for you. The warning becomes an error if you use the "strict" compilation option.

Warning for Hidden Virtual Function

In the following example, the function **D::f(int)** hides the virtual **f()** declared in the base class **B**. The Release 2.0 translator will issue a warning in this case.

```
class B {
        // .....
        virtual void f(double);
        virtual void f(class B*);
        virtual void f();
        // ....
};
class D : public B {
        // ....
        void f(int);          //warning; hides virtual B::f()
        // ....
};
```

Release 1.2 flagged the above declaration of **D::f(int)** as a type mismatch error.

Use of :: Instead of . for Scoping

The use of the period (.) to indicate scoping, as in **C.f()**, is archaic. It was mentioned only as an anachronism in *The C++ Programming Language* (and not at all in the *Reference Manual*). Use of this notation now produces a warning:

```
"", line 2: warning: "." used for qualification; please use "::"
```

The scoping operator . has been changed to :: because of some subtle potential ambiguities and a consistent pattern of misunderstanding of the use of . and :: in expressions.

Inlines May Be Ignored If Too Complicated

In C++, you may specify **inline** for any function; the keyword is just a suggestion to the translator, which the translator can ignore. However, the translator sometimes gives a "sorry, not implemented" error message when it cannot deal with an inline.

Pointers to Class Member Functions

In Release 2.0 the pointer to member operators ->* and .* are considered single tokens, which cannot contain white space.

```
class X {
public:
        int mf(int i) { /* ... */ }
};
typedef int (X::*PMFI)(int);
X xobj;
X *p_xobj = &xobj;

        PMFI pmf = &X::mf;
        int k = (p_xobj->*pmf)(1);
        k = (xobj.*pmf)(1);
        k = (p_xobj -> *pmf)(1);    // 2.0 syntax error
        k = (xobj . *pmf)(1);       // 2.0 syntax error
```

Error for Function Not Returning Class Object As Declared

Declaring a function to return a value, and then defining that function without returning a value produces an error message in both Releases 1.2 and 2.0. However, if the function is declared to return a class object and the function definition does not return anything, Release 2.0 will generate an error message.

Warning for Unspecified Access for Base Class

The Release 2.0 translator produces warnings when the access protection of a base class is unspecified in the declaration of a derived class.

```
class X { int i; public: f(); };

class Y : X { int i; };

g(Y* p) { p->f(); }
```

The call to the function **f()** is illegal because although **f()** is a public member function of **X**, **X** is by default a private base class of **Y**. The declaration of class **Y** above will yield the following warning in Release 2.0:

```
"", line 2: warning:  base X private by default: please be explicit ": private X"
```

Declaring **X** explicitly as a private base class of **Y**:

```
class Y : private X { int i; };
```

eliminates the warning.

Library Changes

The following sections describe changes to the C++ function libraries.

Building Libraries

The following libraries are provided with the C++ Language System:

- the standard C++ library **libC.a**, which includes the iostream library and other functions
- the task library
- the complex library
- the old stream library

The specialized task and complex libraries have been broken out from the basic **libC.a** C++ library, and must now be compiled separately from the translator itself. Additional **make** targets, **libtask.a** and **libcomplex.a**, have been added to the **makefile** for building these libraries.

Information on building these libraries is provided in Chapter 3.

The Iostream Library

The I/O portion of **libC.a** has been significantly extended and re-engineered for Release 2.0. The new implementation is referred to as the **iostream** library. The **iostream** library provides the following enhancements over the old **stream** library:

- greater support for formatting control and operations (there should be no need to revert to **stdio** for specialized formatting)
- support for the full range of UNIX system file operations, including repositioning (**seek**) operations
- redesign of the **streambuf** class to support extensibility
- bug fixes

The new package is largely source compatible with the old unless your program has used the internals of the **streambuf** class. New programs using I/O should **#include <iostream.h>**. A new version of **stream.h** is present for compatibility with the old stream library. Complete information on the iostream library is provided in Chapter 3 of the C++ *Language System Library Manual* and in the manual pages in Appendix A of that volume.

The old I/O library, referred to as the old **stream** library, is still available as **libOstream.a**. If you want to continue using the old **stream** implementation, you must build it explicitly and change any **#include** directives to include **Ostream.h** rather than **stream.h.**

The section called "Converting from Streams to Iostreams" in Chapter 3 of the C++ *Language System Library Manual* details changes you might need to make to convert code that used the old stream library to use the iostream library. Some highlights are listed here.

- The internals of the **streambuf** class in the old stream library were public; in the iostream library they are private, and they are different. Programs that rely on the internals of the old **streambuf** class will need to be changed to run under iostreams in Release 2.0.
- In the old stream library, **streambufs** used character arrays by default as sources and sinks of characters. While the iostream library allows this behavior for compatibility, it is considered obsolete. Instead, the class **strstreambuf** is provided (**strstream.h**) to support incore formatting. Uses of **streambuf, istream,** and **ostream** constructors that take character array arguments should be replaced with uses of **strstreambuf, istrstream,** and **ostrstream** constructors.
- Programs using the **filebuf** class will need to include **fstream.h**. The old stream library provided **istream** and **ostream** constructors that took file descriptor arguments. Uses of these constructors will need to be replaced with uses of the **ifstream** and **ofstream** constructors under iostreams.
- Interactions with the C stdio library differ in the iostream library from the old streams behavior. The iostream library provides classes **stdiostream** and **stdiobuf** to handle such interactions. Any code that passes stdio **FILE** pointers to **filebufs, istreams,** or **ostreams,** should be rewritten to use **stdiobufs** or **stdiostreams** instead.
- In the old stream library, the predefined streams **cin, cout**, and **cerr** were directly connected to the stdio **FILEs stdin, stdout,** and **stderr**. Puts and gets to these streams were done one character at a time. In iostreams, the predefined streams are connected to file descriptors. Users mixing code that uses stdio with code using iostreams should use the function **ios::sync_with_stdio()**. This causes the predefined streams to be connected to the corresponding stdio files in a unit-buffered mode. This means that each inserter will flush characters to output.

- The old stream library provided a **form()** function to allow **printf()**-like formatting. This function is not available in iostreams. Users should use the new formatting flags and manipulators instead. See Chapter 3 of the C++ *Language System Library Manual* and the **ios** manual page for details.
- The old stream library allowed assignment of one stream to another. Such assignments should be changed to use pointers or references to streams in iostreams.
- The iostream library supplies a character inserter (**operator <<(char)**). The old stream library converted **chars** to decimal on output. Code depending on this behavior will need to use explicit casts.

The Task Library

The task library has also been enhanced and re-engineered for Release 2.0. Enhancements include:

- Addition of facilities for tasks to wait for external events, using UNIX System signals.
- AT&T 6386 WGS port
- Sun-2, Sun-3 ports
- Bug fixes

The task library continues to be supplied for AT&T 3B computers and DEC VAX computers. In addition, porting information has been supplied for additional machines. A detailed discussion of the library can be found in Chapter 2 of the C++ *Language System Library Manual.*

A Appendix A

Known Problems

There remain problems in the following general areas of the C++ Language System:

- some declarations, particularly name reuse for types across scopes, are more constrained than the language definition allows
- illegal, but obscure usages are not always detected. These sometimes result in complaints from the underlying C compiler.
- some incompatibilities with the draft ANSI C standard remain, particularly with regard to certain forms of declaration
- some complicated conversions are invalidly rejected
- there are occasional failures to recover after detecting an invalid construction
- there are some instances of spurious warning messages

The following sections detail specific problem areas that remain in the C++ Language System.

Preprocessor Problems

The C++ Language System does not include a version of **cpp** but instead uses the **cpp** resident on the machine. Many **cpps** do not recognize C++ comments. This can sometimes lead to surprising results.

C++-style comments (//) in a macro definition will not be ignored:

```
#include <stream.h>
#define a 5  // define something

main() {
      cout << a;
}
```

The comment in the second macro definition results in the following error messages:

```
error: ';' missing after statement
error:  syntax error
```

Similarly, use of a macro name within a // comment:

```
#include <generic.h>

main() {
      int a; // declare variables
      float f;
}
```

will send many preprocessors into an infinite loop expanding the macro **declare** that is defined inside generic.h. Note that **generic.h** may be included by other files as well. **stream.h**, for example, contains an #include of **generic.h**.

Finally, interactions with normal C comments and C++ comments should be noted. For example:

```
#include <stddef.h>
main() {
      //** this looks like a c-style block comment to cpp
      char *c = NULL;
      /* this is a c-style block comment */
}
```

will result in the following error message if cpp doesn't properly handle C++-style comments:

```
error:  NULL undefined
```

Problems with Code Generation

- The calculation for a pointer to data member for a nested anonymous union is incorrect and will result in bad code at runtime when used. For example:

  ```
  #include <stream.h>

  struct s {
        int ss;
        union {
              int i;
              int *p;
        };
  };

  typedef int s::*p_s_mem;

  main() {
        s S;
        S.ss = 1;
        S.i = 2;
        p_s_mem mem = &s::i;
        p_s_mem mem2 = &s::ss;
        if (mem2 == mem)
        cout << "something's wrong: mem2 == mem0;
  }
  ```

  will detect that the pointers to **i** and to **ss** are equal, which is incorrect. This case failed in Release 1.2 as well.

Restrictions on Global Structures

- In K&R C and the draft ANSI C standard, implementations are free to decide how to treat multiple, uninitialized definitions at global scope. In C++ exactly one definition, initialized or uninitialized, may occur in any compilation.

  In order to enforce this rule, the C++ Language System initializes all global variables to 0. This has the side-effect of generating code that most K&R C implementations will reject when the first element of a **class** or **struct** is a **union**. For example:

  ```
  struct {
        union {
              int i;
        };
  } y;
  ```

  will generate C code including an initialization of **y** that looks like this:

  ```
  struct __C1 y = { 0 } ;
  ```

  This may result in a C compiler error message about incompatible types on the assignment.

- As described above, the C++ Language System ordinarily forces an initialization of all global variables. In the case of static class members this is not currently implemented. This may lead to linkage problems with some linkers. The problem is that some linkers do not pull in modules from an archive if there are no initialized external references. If a file exists whose only external dependency is a static class member, these linkers will fail to include the module with a resulting failure at runtime. For example:

  1. File **ab.h** defines a class with a **static** member:

     ```
     struct A {
           int i;
           A() { i = 3; }
     };

     struct B {
           static A a;
     };
     ```

  2. File **ab.c** defines this symbol:

     ```
     #include "ab.h"

           A B::a;
     ```

  3. File **main.c** refers to it:

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

main()
{
        printf ("%d\n", B::a.i);
}
```

If these files are directly compiled and linked, the expected output of 3 will be printed on the standard output. However, if the file **ab.c** is compiled and stored in a library and linked with main.c then those linkers that do not resolve uninitialized data will print "0."

Restrictions on Typename Reuse

- If a nested class name reuses a member name following the declaration of that member, subsequent uses of the member name following the declaration of the nested class will be misparsed. For example:

```
class class_a{

        class class_1{

        public:
                float memb;
                struct memb{
                        int i;
                        memb(int ii) : i(ii){};
                };
                class_1(int value) : memb(value){
                };
        };
};
```

  Moving the definition of **struct memb** so that it does not fall between the declaration and use of class_a::**memb** resolves the problem.

- For compatibility with existing C programs, the following code fragment should be acceptable. However, the C++ Language System incorrectly rejects this program:

```
struct s { /*...*/ };
f() {
        int s(s*);
        s(0);
}
```

  Inserting a **struct** modifier on the argument declaration for the function **s** allows the code to compile.

typedef Restrictions

- Use of a typedef name in a class header is not recognized:

```
struct B { };

typedef B alias;

struct D : public alias { };

D::D(int i) : alias(i) { };
```

This code results in the following errors:

```
error: unexpected argument list: no base class alias
error:  argument  1 of type int  expected for B::B()
```

Restrictions on Order of Names in Class Template

- The C++ language places no restriction on order of declaration of names within a class template, but some uses before declaration are not accepted. For example:

```
struct foo {
        foo(void (foo::*pmf)(int) = &foo::bar);
        void bar(int);
};
```

generates the following error message:

```
error:  qualified name foo :: bar not found in  foo
```

Restrictions on Resolution of Overloaded Arguments

- The C++ Language System sometimes incorrectly detects an ambiguity. In the following example, it is unable to detect that converting the **pair** variable **p** to **int** and then calling the **complex operator+** on **int** is preferable to converting **p** to **int**, using the standard conversion to **double**, and then invoking **operator+** on **double**:

```
struct complex {
      complex operator+(int);
      complex operator+(double);
};

struct pair {
      operator int();
};

main()
{
      pair p;
      complex x;
      x = x + p;
}
```

This code produces the error message:

```
error: ambiguous argument for complex::operator +():
       complex complex::(int ) and  complex complex::(double )
```

- In this example, the C++ Language System is unable to recognize that **foo->double** is better than **foo->double->char**:

```
struct ostream {
      ostream& operator<<(char);
      ostream& operator<<(unsigned char);
      ostream& operator<<(double);
};

struct foo {
      operator double() const;
};

extern foo f;
extern ostream cout;

main()
{
      cout << f;
}
```

- The C++ Language System fails to accept use of the compiler generated copy constructor for arguments to new: For example:

```
struct foo {
      int i;
      foo(int);
};

main()
{
      foo f(2);
      foo g(f);
      g = foo(f);
      foo* h = new foo(f);
}
```

This code results in the following error message:

```
error: bad argument 1 type for foo::foo(): foo (int expected)
```

for the initialization of **h**. Instead, as with the previous initialization of **g**, the compiler should have generated a **foo(const foo&)** constructor to pass as the argument to **new**.

Other Restrictions

- Use of private and public labels within a named union are not permitted.
- In a **? :** expression, the C++ Language System will incorrectly detect a type clash between the name of a function and a cast of a value to a pointer to the same function type. Explicitly taking the address of the function or reordering the expression allows the code to compile:

```
typedef void      *(*MakeFn)(void *);

MakeFn fnp;

extern void *g_mknode(void *attrs);

void foo(int insert)
{
      fnp = (insert ? g_mknode : (MakeFn)0);   // fails
      fnp = (insert ? &g_mknode : (MakeFn)0); // compiles
      fnp = (insert ? (MakeFn)0 : g_mknode);   // compiles
}
```

Failure to Detect Illegal Code

- Arrays of objects with constructors may only be declared for objects with a default constructor, that is, a constructor with no arguments. This is not the same as a constructor that has all of its arguments declared with defaults. The C++ Language System fails to enforce this restriction when the array is itself a member of another class. For example, the following declarations are accepted, but the resulting code does not pass the default arguments to the constructor:

```
struct s {
      int ss;
      s(int a =2, int b =3);
};

s::s(int a, int b) {
      ss = a * b;
}
struct t {
      s ses[2];
};
```

- With Release 2.0, static class members may be of any type, but must be explicitly initialized exactly once in the compilation. However, the system does not detect static members that are multiply defined. Static class members that have constructors will be initialized once for each file in which the static member is defined. For example:

  File **a.h** defines a class with a static member:

```
#include <stream.h>
struct A {A() {cout << "A::A()0;};};

struct B {static A ab;};
```

  and file **a.c** defines the class static:

```
#include "a.h"
A B::ab;
```

  as does file **main.c**:

```
#include "a.h"
A B::ab;
main() {
};
```

  The result will be two calls to the **A** constructor for the static member **ab**.

- Protection is not honored in an aggregate initialization when the private datum is hidden by an extra level of indirection. For example, the following illegal initialization will be accepted:

```
class A {
      int a;
};

struct B {
      A obj;
};

B object = { 5 };
```

- Members of **const** objects are not recognized to be **const** themselves. For example:

```
struct T { int a; };

void f()
{
      const T t;

      t.a = 0;
      *&t.a = 0;
}
```

Only the first assignment will be correctly flagged as an error. Similarly, the C++ Language System will fail to detect the illegal call of the non-const member function **Foo::foo** from the **const** member **Bar::bop**:

```
class Foo {
      int x;
public:
      void foo(int a) {x = a;}
      };

class Bar {
      Foo p;
public:
      void bop() const {p.foo(2);}
      };
```

- The **overload** keyword is incorrectly accepted as a variable modifier. Generally, this is harmless and the duplicate declaration is detected. However, when two variables are overloaded and one is of type **int**, the **int** declaration is quietly ignored:

```
overload int i;
overload float i; // no complaint, i is float

overload int k;
overload float k;
overload double k;      // error detected
```

- A meaningless and illegal declaration of constructors and virtuals as **friends** is accepted.
- A meaningless and illegal use of the **const** qualifier on the declaration of a global function or a static member function is accepted.
- A meaningless and illegal default argument for a function defined to take a **void** is accepted.
- A meaningless and illegal use of the **extern** keyword is not detected within an argument list.

Failure to Detect Illegal Code with Resultant cc Errors

- The Language System accepts aggregate initialization of a **struct** with a union member; however, most K&R C compilers do not accept the output code:

```
union U {
      int i;
      float f;
};

struct S {
      U mem;
};

S obj = { 1 };
```

- The Language System allows **sizeof()** applied to a function or a bitfield, which causes errors in the underlying C compiler.

Failure to Detect Illegal Code with Resultant Indeterminate Effect

- The translator fails to detect illegal use of a member whose name reuses that of a type before the name has been declared to be a member. Depending on the context, this may yield errors from the C++ Language System, errors from your underlying C compiler, or code that is likely not to match the programmer's intent.

```
typedef int Integer ;

class A {
public:
        int minus3() { return ( Integer ) - 3; }
        int Integer;
} ;

class B {
public:
        int Integer;
        int minus3() { return ( Integer ) - 3; }
} ;
```

In class **A**, **Integer** is taken as the **typedef** and results in a cast of -3 to **int**; in class **B**, it is parsed as the **int** member named **Integer**.

Incompatibilities with the Draft ANSI C Standard

- Fails to accept a draft ANSI conforming declaration for nested initialization of **enums**:

  ```
  enum e { e1 = (enum { z = 10 } ) 3, e2};
  ```

  This code produces the following error:

  ```
  error: cast to non-integral type in constant expression
  ```

- The translator fails to accept draft ANSI conforming declarations for functions taking function arguments:

  ```
  void f(int ());
  ```

  This declaration produces the following error:

  ```
  error:  argument type expected for f()
  ```

  A similar declaration taking a function pointer argument is accepted:

  ```
  void f(int(*)());
  ```

Spurious Warnings

- Some instances of "used but not set" warnings are invalid. For example:

  ```
  struct A {
        operator =(int);
        operator int();
  };
  main() {
        A a;
        int i;
        a = 5;
        i = a;
  }
  ```

  falsely warns about **a**.

- Some instances of non-const reference (&) argument warnings are invalid. For example:

  ```
  struct foo {
        foo();
        foo(int);
  };


  void faz(foo&);

  main()
  {
        faz(foo(3));
  }
  ```

  falsely warns about **foo**.

- The C++ Language System is sometimes too conservative in deciding when it is necessary to call destructors. On some C compilers this results in spurious warnings about unreachable code. For example:

```
struct A {
      A();
      ~A();
};
main() {
      extern int i;
      switch(i) {
            case 0: {
            A avar;
            break;
                                                                        };
                                                                  };
};
```

may result in an invalid C compiler warning.

Similarly, for the following case, destructors are properly called on each return path but also from the end of the function:

```
struct A {
      A();
      ~A();
};
main() {
      A avar;
      extern int i;
      if (i)
            return i;
      else
            return i;
}
```

Error Recovery Failures

- Errors in class declarations occasionally lead to cases from which the translator is unable to recover. For example, the following error is correctly detected, but compilation stops prematurely:

```
      class a {
            class a {         // error class a redefined
            };
      };
```

This code produces the error message:

```
error:  class a  redefined
sorry, cannot recover from previous error
```

B Appendix B

Documentation

The AT&T C++ Language System includes four documents:

- *AT&T C++ Language System Release Notes* (select code 307-090) – this document, which describes the C++ Language System and programming language, the contents of Release 2.0 of the Language System, how to install and port the C++ translator, and compatibility between this release and previous releases.
- *AT&T C++ Language System Product Reference Manual* (select code 307-146) – describes the C++ programming language in detail.
- *AT&T C++ Language System Selected Readings* (select code 307-144) – provides papers describing aspects of the C++ programming language, including a discussion of new features, a tutorial introduction to the language, and other miscellaneous chapters on significant language features.
- *AT&T C++ Language System Library Manual* (select code 307-145) – describes the three class libraries provided with the C++ Language System:
  - the complex arithmetic library
  - the task library
  - the iostream library

These documents may also be ordered separately (see "How to Order Documents").

Many other documents containing related material about the programming environment on the UNIX system may be purchased. They are listed in the *AT&T Computer Systems Documentation Catalog* (select code 300-000). The most important documents available are described below:

- *The C++ Programming Language* by Bjarne Stroustrup (select code 320-025)

  This is the standard introduction to the C++ programming language. It includes discussions of all the basic features of the language.
- *UNIX System V User's Guide* (select code 307-231)

  The *User's Guide* introduces the basic principles of the UNIX operating system. It includes tutorials on using the UNIX system text editors **ed** and **vi**, shell programming, and communication tools.
- *UNIX System V User's Reference Manual* (select code 307-012)

  The *User's Reference Manual* contains reference material in the form of manual pages for UNIX system commands for general users.
- *UNIX System V Programmer's Reference Manual* (select code 307-013)

  The *Programmer's Reference Manual* contains reference material in the form of manual pages for all commands used by UNIX system programmers, including standard system calls, subroutines, libraries, file formats, macro packages, and character-set tables.

NOTE The UNIX system is portable to many different computers. The reference manuals listed for the UNIX system are available for many of these computers. The manuals listed above are usually included with the computers; if you need additional copies, refer to the section "How to Order Documents" for ordering information.

- *UNIX System V Programmer's Guide* (Select Code 308-139)

  The *Programmer's Guide* contains descriptive information about the C language and associated libraries, the C compiler, the C program checker (**lint**), the symbolic debugging program (**sdb**), and many other related topics. It includes a summary of the grammar and rules of the C programming language.

- *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (select code 307-136)

  This is the standard introduction to the C programming language. It includes a tutorial introduction to C, complete discussions of all the major features of the language, and a reference section.

- *C Programmer's Handbook* (select code 320-022)

  The *C Programmer's Handbook* is a short reference guide to the C programming language. Topics covered include syntax, data types, operators and expressions, statements, functions, declarations, program structure, libraries, formatted input/output, and portable C programs.

- *The UNIX Programming Environment* by Brian W. Kernighan and Rob Pike (select code 311-026)

  This is an introduction to the UNIX operating system. It includes a tutorial introduction followed by information on the file system, shell programming, system calls, document preparation, and other topics.

- *AT&T 3B2 Computer C Programming Language Utilities/Advanced Programming Utilities Product Overview* (select code 308-227)

  The *Product Overview* contains a summary of the C Programming Language Utilities and Advanced Programming Utilities and a brief technical description of most of both products' features. It is especially helpful for new users.

The following textbooks on the C++ programming language are also available:

- *A C++ Primer* by Stanley Lippman (1989, Addison-Wesley, Reading, MA)
- *Programming in* C++ by Stephen C. Dewhurst and Kathy T. Stark (1989, Prentice-Hall, Englewood Cliffs, NJ)

How to Order Documents

Order Documents

CALL		WRITE
in continental USA:	1-800-432-6600	AT&T Customer Information Center
in Canada:	1-800-255-1242	Customer Service Representative
elsewhere:	1-317-352-8556	P.O. Box 19901
		Indianapolis, Indiana 46219

Sign up for UNIX system or 3B2 Computer courses

CALL		TELEX
in continental USA:	1-800-247-1212	919-365-ATTI-UR
elsewhere:	1-201-953-7554	Attention: Training Registration

Find out about UNIX system source licenses

CALL		WRITE
in continental USA		Software Licensing
(except NC):	1-800-828-UNIX	Guilford Center
		P.O. Box 25000
NC and elsewhere:	1-919-279-3666	Greensboro, NC 27420

Contact marketing representative about AT&T hardware and software

CALL		WRITE
in continental USA:	1-800-247-1212	not applicable
elsewhere:	1-201-953-7554	

Other Sources

The *AT&T Computer Systems Documentation Catalog* (select code 300-000) lists documents for all models of the AT&T 3B Computer family, and for hardware add-on products, AT&T software, and releases of UNIX System V.

The *AT&T Computer Software Catalog* (select code 311-020) lists over 500 software products reviewed or certified for compatibility with AT&T computers.

I Index

Index