Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

А. В. Сахарук

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В СИСТЕМАХ УПРАВЛЕНИЯ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
по выполнению курсовой работы
для студентов специальности 1-53 01 07
«Информационные технологии и управление
в технических системах»
дневной формы обучения

Гомель 2025

УДК 004.415.2(075.8)
ББК 32.972.1я73
С22

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 9 от 17.05.2023 г.*)

Рецензент: доц. каф. «Информационные технологии» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *В. С. Захаренко*

Сахарук, А. В.

С22 Объектно-ориентированное программирование в системах управления : учеб.-метод. пособие по выполнению курсовой работы для студентов специальности 1-53 01 07 «Информационные технологии и управление в технических системах» днев. формы обучения / А. В. Сахарук. – Гомель : ГГТУ им. П. О. Сухого, 2025. – 139 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Содержит основные положения для выполнения курсовой работы, теоретические и практические сведения по работе с компилятором и сборке проектов, основы проектирования классов, приемы работы с файлами, обработки исключительных ситуаций и проектирования пользовательского интерфейса.

УДК 004.415.2(075.8)
ББК 32.972.1я73

ВВЕДЕНИЕ

Данное учебно-методический пособие предназначено для студентов, изучающих курс «Объектно-ориентированное программирование в системах управления». В ходе изучения данного курса студенты ознакомятся с основными принципами объектно-ориентированного программирования и научатся применять их на практике при разработке программ.

Цель данного курсового проекта заключается в том, чтобы научить студентов разрабатывать программы, используя язык программирования C++. Проект предполагает выполнение задач, связанных с объектно-ориентированным подходом:, создание классов, наследование, полиморфизм и инкапсуляция.

Студенты, успешно выполнившие данный курсовой проект, научатся:

– Понимать основные концепции объектно-ориентированного программирования и применять их в практической разработке на C++.

– Проектировать и разрабатывать классы и их взаимодействие в программах.

– Применять принципы наследования и полиморфизма для создания гибких и расширяемых программных систем.

– Использовать инкапсуляцию для обеспечения безопасности и модульности кода.

– Разрабатывать эффективные и оптимизированные программные решения на языке C++.

Выполнение курсового проекта представляет собой важную часть обучения студентов, позволяя им применить полученные теоретические знания на практике и развить навыки программирования в объектно-ориентированной парадигме. В ходе работы над проектом студенты также будут сталкиваться с задачами планирования, анализа требований, тестирования и документирования, что является неотъемлемой частью процесса разработки программных систем.

1 Общие положения

1.1 Задание по курсовому проектированию

В соответствии с индивидуальным заданием разрабатывается программное обеспечение на языке C++ с использованием компилятора GCC.

Расчетно-пояснительная записка объемом 25–30 страниц должна содержать:

– титульный лист;

– задание на курсовую работу;

– содержание;

– введение;

– программная часть;

– заключение;

– список литературы;

– приложения.

– введение объемом 5–10 страниц должно содержать краткое изложение теоретических сведений по тематике курсовой работы.

Программная часть содержит разделы, оглавление и количество которых соответствует заданию на курсовое проектирование. А так же блоксхемы алгоритмов.

В заключении приводятся основные выводы и результаты, полученные в процессе проектирования.

В приложении проводится полный исходный код разрабатываемой программы с разделением по файлам, а также сценарий сборки приложения.

Графической частью курсовой работы является чертеж блоксхемы алгоритма работы программы, выполненный на плоттере на листе ватмана стандартной формы формата А1 согласно ГОСТ.

Выполнение чертежа А1, заполнение основной надписи на чертежах должны соответствовать правилам оформления дипломного проекта [1].

1.2 Порядок выполнения курсовой работы

В самом начале студент должен изучить индивидуальное задание и определить предметную область, для которой разрабатывается программное обеспечение.

Создать дерево исходного кода и сценарий сборки приложения в соответствии с разделом 2 (Компиляция и сборка проектов). Код приложения должен быть логически разделен на части, которые будут

располагаться в отдельных файлах исходного кода в сопровождении с соответствующими заголовочными файлами, в которых будут прописаны прототипы функций и структуры классов. Каждый класс должен иметь отдельный набор состоящий из заголовочного файла и файла с исходным кодом.

Затем, соответствии с заданием и предметной областью, определить перечень объектов, для описания которых необходимо создать типы данных (раздел 5. Работа с классами). Реализовать методы, позволяющие обрабатывать данные. Объемные данные должны располагаться в динамической области памяти (раздел 3. Работа с указателями и динамической памятью). Так же в пользовательских типах данных должны быть предусмотрены перегрузки стандартных операций, которые необходимы для обработки данных (раздел 6. Перегрузка операций). Для длительного сохранения файлов необходимо применять файлы в файловой системе (раздел 4. Работа с файлами).

Разрабатываемое программное обеспечение должно быть оснащено алгоритмами для обработки ошибок и исключительных ситуаций (соответствии с разделом 7. Обработка исключительных ситуаций) для обеспечения стабильной работы и исключения возможности потери данных.

Для взаимодействия с пользователем необходимо разработать пользовательский интерфейс. В зависимости от задания, он может быть основан как на обычном тексте (раздел 9. Потоковый ввод-вывод), так и на символах псевдографики (раздел 8. Построение псевдографического интерфейса).

После реализации программного обеспечения необходимо смоделировать различные режимы его использования и провести комплексное тестирование. В случае если в процессе тестирования будут выявлены ошибки и сбои в работе программного обеспечения, необходимо определить причины и устранить их.

После выполнения технических работ необходимо оформить пояснительную записку в соответствии с пунктом 1.1 и техническим заданием.

2 Компиляция и сборка проектов

2.1 Основные понятия

Компиляция – трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера). Входной информацией для компилятора (исходный код) является описание алгоритма или программа на объектно-ориентированном языке, а на выходе компилятора – эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

Компоновщик (также редактор связей или ли́нкер, отангл. *link editor, linker*) – это инструментальная программа, которая производит компоновку (*«линковку»*): принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль.

GNU Compiler Collection (обычно используется сокращение GCC) – набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU. GCC является свободным программным обеспечением, распространяется фондом свободного программного обеспечения (FSF) на условиях GNU GPL и GNU LGPL и является ключевым компонентом GNU toolchain. Он используется как стандартный компилятор для свободных UNIX-подобных операционных систем.

Изначально названный GNU C Compiler поддерживал только язык Си. Позднее GCC был расширен для компиляции исходных кодов на таких языках программирования, как C++, Objective-C, Java,Фортран, Ada и Go.

Программа **gcc**, запускаемая из командной строки, представляет собой надстройку над группой компиляторов. В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, **gcc** запускает необходимые препроцессоры, компиляторы, линкеры.

Файлы с расширением **.cc** или **.C** рассматриваются, как файлы на языке C++, файлы с расширением **.c** как программы на языке C, а файлы с расширением **.o** считаются объектными.

Чтобы откомпилировать исходный код C++, находящийся в файле **F.cc**, и создать объектный файл **F.o**, необходимо выполнить команду:

gcc -c F.cc

Опция **– c** означает «только компиляция».

Чтобы скомпоновать один или несколько объектных файлов, полученных из исходного кода – **F1.o**, **F2.o**, ... – в единый исполняемый файл **F**, необходимо ввести команду:

gcc -o F F1.o F2.o

Опция **-o** задает имя исполняемого файла.

2.2 Опции компиляции

Основные опции компиляции приведены в таблице 2.1.

Таблица 2.1 – Основные опции компиляции

Опция	**Назначение**
-c	Эта опция означает, что необходима только компиляция. Из исходных файлов программы создаются объектные файлы в виде name.o. Компоновка не производится
-Dname=value	Определить имя name в компилируемой программе, как значение value. Эффект такой же, как наличие строки #define name value в начале программы. Часть =value может быть опущена, в этом случае значение по умолчанию равно 1
-o file-name	Использовать file-name в качестве имени для создаваемого файла
-lname	Использовать при компоновке библиотеку libname.so
-Llib-path -Iinclude-path	Добавить к стандартным каталогам поиска библиотек и заголовочных файлов пути lib-path и include-path соответственно
-g	Поместить в объектный или исполняемый файл отладочную информацию для отладчика gdb. Опция должна быть указана и для компиляции, и для компоновки. В сочетании -g рекомендуется использовать опцию отключения оптимизации –O0
-MM	Вывести зависимости от заголовочных файлов, используемых в Си или C++ программе, в формате, подходящем для утилиты make. Объектные или исполняемые файлы не создаются

Окончание таблицы 2.1

-pg	Поместить в объектный или исполняемый файл инструкции профилирования для генерации информации, используемой утилитой gprof. Опция должна быть указана и для компиляции, и для компоновки. Собранная с опцией -pg программа при запуске генерирует файл статистики. Программа gprof на основе этого файла создает расшифровку, указывающую время, потраченное на выполнение каждой функции
-Wall	Вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы
-O1 -O2 -O3	Различные уровни оптимизации
-O0	Не оптимизировать. Если вы используете многочисленные -O опции с номерами или без номеров уровня, действительной является последняя такая опция
-I	Используется для добавления ваших собственных каталогов для поиска заголовочных файлов в процессе сборки
-L	Передается компоновщику. Используется для добавления ваших собственных каталогов для поиска библиотек в процессе сборки
-l	Передается компоновщику. Используется для добавления ваших собственных библиотек для поиска в процессе сборки

Рассмотри пример сборки проекта с необходимыми опциями.

g++ main.cpp -lncurses -lpanel -o main

В данном примере собирается программа, исходный код которой содержится в файле main.cpp, имя исполняемого файла, который необходимо получить main, при этом в проекте используются две библиотеки: libncurses и libpanel.

2.3 Утилита make

make – утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исход-

ного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.

Общий формат команды:
make [-f make-файл] [цель] ...]

Файл ищется в текущем каталоге. Если ключ -f не указан, используется имя по умолчанию для make-файла – Makefile (однако, в разных реализациях make кроме этого могут проверяться и другие файлы, например GNUmakefile).

make открывает make-файл, считывает правила и выполняет команды, необходимые для создания указанной *цели*.

Стандартные цели для сборки дистрибутивов GNU:

– all – выполнить сборку пакета;

– install – установить пакет из дистрибутива (производит копирование исполняемых файлов, библиотек и документации в системные каталоги);

– uninstall – удалить пакет (производит удаление исполняемых файлов и библиотек из системных каталогов);

– clean – очистить дистрибутив (удалить из дистрибутива объектные и исполняемые файлы, созданные в процессе компиляции);

– distclean – очистить все созданные при компиляции файлы и все вспомогательные файлы, созданные утилитой ./configure в процессе настройки параметров компиляции дистрибутива.

Помимо целей, у данной утилиты есть ряд опций, которые позволяют управлять процессом сборки. Перечень основных пций приведена в таблице 2.2.

Таблица 2.2 – Основные опции утилиты make

Опция	**Описание**
-f <имя_файла>	Указывает make использовать указанный файл вместо стандартного имени Makefile или makefile

Окончание таблицы 2.2

-n или --just-print	Показывает, какие команды будут выполнены, но не выполняет их на самом деле. Это полезно для проверки правильности сценария сборки без реального запуска команд
-B или --always-make	Заставляет make выполнить цели, даже если файлы не были изменены. Это полезно, когда вы хотите пересобрать все, несмотря на изменения
-j <количество_параллельных_задач>	Указывает make выполнять несколько задач параллельно. Это может значительно ускорить процесс сборки, особенно на многоядерных системах
-k или --keep-going	Позволяет make продолжать выполнение команд, даже если одна из них завершается с ошибкой. Это полезно, когда вам нужно выполнить максимальное количество задач, несмотря на ошибки
--dry-run	Показывает, какие команды будут выполнены, но не выполняет их на самом деле. Отличается от опции -n тем, что также показывает зависимости между файлами
-C <каталог>	Переходит в указанный каталог перед выполнением команд. Это полезно, когда ваш Makefile ожидает, что текущий каталог будет установлен определенным образом
-s или --silent	Подавляет вывод команд, выполняемых make. Это полезно, когда вы хотите сократить вывод и сделать его более читабельным
-e или --environment-overrides	Позволяет переменным окружения переопределять переменные, определенные в Makefile
-t или --touch	Обновляет временные метки файлов, чтобы они выглядели так, как будто были изменены. Это может быть полезно для заставления make пересобрать файлы, даже если они не были изменены

2.4 Make-файл

Простой make-файл состоит из "правил" (rules) следующего вида:

```
цель ... : пререквизит ...
        команда
        ...
        ...
```

Обычно, цель (target) представляет собой имя файла, который генерируется в процессе работы утилиты make. Примером могут служить объектные и исполняемый файлы собираемой программы. Цель также может быть именем некоторого действия, которое нужно выполнить (например, `clean' – очистить).

Пререквизит (prerequisite) – это файл, который используется как исходдные данные для порождения цели. Очень часто цель зависит сразу от нескольких файлов.

Команда – это действие, выполняемое утилитой make. В правиле может содержаться несколько команд – каждая на свое собственной строке. Важное замечание: строки, содержащие команды обязательно должны начинаться с символа табуляции.

Обычно, команды находятся в правилах с пререквизитами и служат для создания файла-цели, если какой-нибудь из пререквизитов был модефицирован. Однако, правило, имеющее команды, не обязательно должно иметь пререквизиты. Например, правило с целью `clean' ("очистка"), содержащее команды удаления, может не иметь пререквизитов.

Правило (rule) описывает, когда и каким образом следует обновлять файлы, указанные в нем в качестве цели. Для создания или обновления цели, make исполняет указанные в правиле команды, используя пререквизиты в качестве исходных данных. Правило также может описывать каким образом должно выполняться некоторое действие.

Помимо правил, make-файл может содержать и другие конструкции, однако, простой make-файл может состоять и из одних лишь правил. Правила могут выглядеть более сложными, чем приведенный выше шаблон, однако все они более или менее соответствуют ему по структуре.

Пример простого make-файла, в котором описывается, что исполняемый файл edit зависит от восьми объектных файлов, которые, в свою очередь, зависят от восьми соответствующих исходных файлов и трех заголовочных файлов.

В данном примере, заголовочный файл `defs.h' включается во все файлы с исходным текстом. Заголовочный файл `command.h' включа-

ется только в те исходные файлы, которые относятся к командам редактирования, а файл `buffer.h' – только в "низкоуровневые" файлы, непосредственно оперирующие буфером редактирования.

```
edit : main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

main.o : main.c defs.h
                cc -c main.c
kbd.o : kbd.c defs.h command.h
                cc -c kbd.c
command.o : command.c defs.h command.h
                cc -c command.c
display.o : display.c defs.h buffer.h
                cc -c display.c
insert.o : insert.c defs.h buffer.h
                cc -c insert.c
search.o : search.c defs.h buffer.h
                cc -c search.c
files.o : files.c defs.h buffer.h command.h
                cc -c files.c
utils.o : utils.c defs.h
                cc -c utils.c
clean :
      rm edit main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

Для повышения удобочитаемости, строки разбиты на две части с помощью символа обратной косой черты, за которым следует перевод строки.

В приведенном примере, целями, в частности, являются объектные файлы `main.o' и `kbd.o', а также исполняемый файл `edit'. К пререквизитам относятся такие файлы, как `main.c' и `defs.h'. Каждый объект-

ный файл, фактически, является одновременно и целью и пререквизитом. Примерами команд могут служить `cc -c main.c' и `cc -c kbd.c'.

В случае, если цель является файлом, этот файл должен быть перекомпилирован или перекомпонован всякий раз, когда был изменен какой-либо из его пререквизитов. Кроме того, любые пререквизиты, которые сами генерируются автоматически, должны быть обновлены первыми. В нашем примере, исполняемый файл `edit' зависит от восьми объектных файлов; объектный файл `main.o' зависит от исходного файла `main.c' и заголовочного файла `defs.h'.

За каждой строкой, содержащей цель и пререквизиты, следует строка с командой. Эти команды указывают, каким образом надо обновлять целевой файл. В начале каждой строки, содержащей команду, должен находится символ табуляции. Именно наличие символа табуляции является признаком, по которому make отличает строки с командами от прочих строк make-файла.

Цель `clean' является не файлом, а именем действия. Поскольку, при обычной сборке программы это действие не требуется, цель `clean' не является пререквизитом какого-либо из правил. Следовательно, make не будет "трогать" это правило, пока не произойдет принудительный вызов. Это правило не только не является пререквизитом, но и само не содержит каких-либо пререквизитов. Таким образом, единственное предназначение данного правила - выполнение указанных в нем команд. Цели, которые являются не файлами, а именами действий называются абстрактными целями (phony targets).

В приведенном выше примере, в правиле для `edit' дважды перечисляется список объектных файлов программы:

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Подобное дублирование чревато ошибками. При добавлении в проект нового объектного файла, можно добавить его в один список и забыть про другой. Можно устранить подобный риск, и, одновременно, упростить make-файл, используя переменные. Переменные (variables) позволяют, один раз определив текстовую строку, затем использовать ее многократно в нужных местах.

Обычной практикой при построении make-файлов является использование переменной с именем objects, OBJECTS, objs, OBJS, obj,

или OBJ, которая содержит список всех объектных файлов программы. Можно определить подобную переменную с именем objects таким образом:

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

Далее, всякий раз, когда нужен будет список объектных файлов, можно использовать значение этой переменной с помощью записи `$(objects)'. Вот как будет выглядеть предыдущий пример с использованием переменной для хранения списка объектных файлов:

```
objects = main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

На самом деле, нет необходимости явного указания команд компиляции отдельно для каждого из исходных файлов. Утилита make сама может "догадаться" об использовании нужных команд, поскольку у нее имеется, так называемое, неявное правило (implicit rule) для обновления файлов с расширением `.o' из файлов с расширеним `.c', с помощью команды `cc -c'. Например, она бы использовала команду `cc -c main.c -o main.o' для преобразования файла `main.c' в файл `main.o'. Таким образом, можно убрать явное указание команд компиляции из правил, описывающих построение объектных файлов.

Когда файл с расширением `.c' автоматически используется подобным образом, он также автоматически добавляется в список пререквизитов "своего" объектного файла. Таким образом, мы вполне можем убрать файлы с расширением `.c' из списков пререквизитов объектных файлов.

Пример можно модифицировать следующим образом:

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

Примерно так и выглядят make-файлы в реальной практике. Если для создания объектных файлов используются только неявные правила, то можно использовать другой стиль написания make-файлов.

В таком make-файле записи группируются по их пререквизитам, а не по их целям. Вот как может выглядеть подобный make-файл:

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Здесь, заголовочный файл `defs.h' объявляется пререквизитом для всех объектных файлов программы. Файлы `command.h' и `buffer.h' являются пререквизитами для перечисленных объектных файлов.

Часто, в make-файле указывается, каким образом можно выполнить некоторые другие действия, не относящиеся к компиляции программы. Таким действием, например, может быть удаление все объектных и исполняемых файлов программы для очистки каталога. Вот как можно было бы написать правило для очистки каталога:

```
clean:
        rm edit $(objects)
```

Однако практике, это правило реалиуется чуть более сложным способом, предполагающим возможность непредвиденных ситуаций:

```
.PHONY : clean
clean :
       -rm edit $(objects)
```

Такая запись предотвратит возможную путаницу если, вдруг, в каталоге будет находится файл с именем `clean', а также позволит make продолжить работу, даже если команда rm завершится с ошибкой.

3 Работа с указателями и динамической памятью

Указатель – это переменная, содержащая адрес переменной. Указатели широко применяются в Си отчасти потому, что в некоторых случаях без них просто не обойтись, а отчасти потому, что программы с ними обычно короче и эффективнее. Указатели и массивы тесно связаны друг с другом: в данной главе мы рассмотрим эту зависимость и покажем, как ею пользоваться. Наряду с goto указатели когда то были объявлены лучшим средством для написания малопонятных программ. Так оно и есть, если ими пользоваться бездумно. Ведь очень легко получить указатель, указывающий на чтонибудь совсем нежелательное.

3.1 Указатели и адреса

Начнем с того, что рассмотрим упрощенную схему организации памяти. Память типичной машины подставляет собой массив последовательно пронумерованных или проадресованных ячеек, с которыми можно работать по отдельности или связными кусками. Применительно к любой машине верны следующие утверждения: один байт может хранить значение типа char, двухбайтовые ячейки могут рассматриваться как целое типа short, а четырехбайтовые как целые типа long. Указатель это группа ячеек (как правило, две или четыре), в которых может храниться адрес. Так, если с имеет тип char, а p указатель на с, то ситуация выглядит следующим образом:

p = &c;

Унарный оператор & выдает адрес объекта, так что инструкция присваивает переменной p адрес ячейки с. Оператор & применяется только к объектам, расположенным в памяти: к переменным и элементам массивов. Его операндом не может быть ни выражение, ни константа, ни регистровая переменная.

Унарный оператор * есть оператор косвенного доступа. Примененный к указателю он выдает объект, на который данный указатель указывает. Предположим, что x и y имеют тип int, а ip – укаэатель на int.

```
int x = 1, y = 2, z[10];
int *ip;    /* ip – указатель на int */
ip = &x;    /* теперь ip указывает на x */
y = *ip;    /* y теперь равен 1 */
 *ip = 0;    /* x теперь равен 0 */
 ip = &z[0]; /* ip теперь указывает на z[0] */
```

Объявления x, y и z нам уже знакомы. Объявление указателя ip int *ip; мы стремились сделать мнемоничным оно гласит: "выражение *ip имеет тип int". Синтаксис объявления переменной "подстраивается" под синтаксис выражений, в которых эта переменная может встретиться. Указанный принцип применим и в объявлениях функций. Например, запись
double *dp, atof (char *); означает, что выражения *dp и atof(s) имеют тип double, а аргумент функции atof есть указатель на char.

Стоит обратить внимание, что указателю разрешено указывать только на объекты определенного типа. Если ip указывает на x целочисленного типа, то *ip можно использовать в любом месте, где допустимо применение x; например, *ip = *ip + 10; увеличивает *ip на 10.

Унарные операторы * и & имеют более высокий приоритет, чем арифметические операторы, так что присваивание y = *ip + 1; берет то, на что указывает ip, и добавляет к нему 1, а результат присваивает переменной y. Аналогично *ip += 1; увеличивает на единицу то, на что указывает ip; те же действия выполняют ++*ip; и (*ip)++;

В последней записи скобки необходимы, поскольку если их не будет, увеличится значение самого указателя, а не то, на что он указывает. Это обусловлено тем, что унарные операторы * и ++ имеют одинаковый приоритет и порядок выполнения справа налево. И, наконец, так как указатели сами являются переменными, в тексте они могут встречаться и без оператора косвенного доступа. Например, если iq есть другой указатель на int, то iq = ip; копирует содержимое ip в iq, чтобы ip и iq указывали на один и тот же объект.

Рассмотрим простейшую программу с указателем на переменную типа int:

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Start programm" << endl;
   int a =0;
   int *b=&a;
   cout << "sizeof(a)=" << sizeof(a) << endl;
   cout << "sizeof(b)=" << sizeof(b) << endl;
   cout << "a=" << a << endl;
   cout << "b=" << b << endl;
```

```
    cout << "*b=" << *b << endl;
    cout << "Increment a" << endl;
    a++;
    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "*b=" << *b << endl;
    cout << "Increment *b" << endl;
    (*b)++;
    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "*b=" << *b << endl;
    cout << "Increment b" << endl;
    b++;
    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "*b=" << *b << endl;
    return 0;
}
```

Сначала подключаем библиотеку iostream для использования потокового ввода/вывода. Опция using namespace std указывает компилятору что по умолчанию используются функции из пространства имен std. Если данная опция не будет указана, то команда вывода информации через cout будет выглядеть следующим образом:

```
std::cout << "a=" << a << std::endl;
```

Таким образом, когда в программе используются функции только из пространства std, наиболее удобно применять данную опцию для уменьшения исходного кода.

Объявляем переменную a типа int и указатель b на тип int. Переменной a присвоим значение 0 а указателю b адрес переменно a:

```
int a =0;
int *b=&a;
```

С помощью функции sizeof() определим объем, который занимают в памяти наша переменная и укащатель. Так как компилятор предназначен для 32-х битной платформы, тип int а так же указатель на него будут занимать в оперативной памяти объем равный 4 байта.

```
cout << "sizeof(a)=" << sizeof(a) << endl;
cout << "sizeof(b)=" << sizeof(b) << endl;
```

Теперь проведем несколько опытов с переменной и указателем. Для начала выведем на экран значение нашей переменной, значение указателя, а так же значение объекта на который указывает указатель:

```
cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;
```

В первом и третьем случае на экране будет отображено значение нашей переменной, которое равно 0. А вот во втором случае будет выведен адрес, который в памяти занимает наша переменная.

Теперь проведем инкремент переменной a и снова выведем значения на экран:

```
a++;
cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;
```

Значении переменно a после операции инкремента увеличиться на 1. А адрес, находящийся в указателе, останется неизменный. Теперь проведм инкремент объекта, на который указывает b:

```
(*b)++;
cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;
```

На экране мы увидим те же изменения как и в первом случае – значение переменной увеличиться на 1, а адрес останется неизменным. А вот теперь выполним инкремент самого указателя:

```
b++;
cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "*b=" << *b << endl;
```

После выполнения данных действий переменная a останется неизменной, а вот адрес сместиться вправо на 4 байта и значение объекта, на который указывает указатель b, не будет совпадать со значением переменной a.

3.2 Указатели и аргументы функций

Поскольку в Си функции в качестве своих аргументов получают значения параметров, нет прямой возможности, находясь в вызванной функции, изменить переменную вызывающей функции. В программе

сортировки нам понадобилась функция swap, меняющая местами два неупорядоченных элемента. Однако недостаточно написать
swap(a, b); где функция swap определена следующим образом:

```
void swap(int x, int y) /* НЕВЕРНО */
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Поскольку swap получает лишь копии переменных a и b, она не может повлиять на переменные a и b той программы, которая к ней обратилась. Чтобы получить желаемый эффект, вызывающей программе надо передать указатели на те значения, которые должны быть изменены: swap(&a, &b);

Так как оператор & получает адрес переменной, &a есть указатель на a. В самой же функции swap параметры должны быть объявлены как указатели, при этом доступ к значениям параметров будет осуществляться косвенно.

```
void swap(int *px, int *py) /* перестановка *px и *py */
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```

Аргументы указатели позволяют функции осуществлять доступ к объектам вызвавшей ее программы и дают возможность изменить эти объекты.

Рассмотрим, например, функцию getint, которая осуществляет ввод в свободном формате одного целого числа и его перевод из текстового представления в значение типа int. Функция getint должна возвращать значение полученного числа или сигнализировать значением EOF о конце файла, если входной поток исчерпан. Эти значения должны возвращаться по разным каналам, так как нельзя рассчитывать на то, что полученное в результате перевода число никогда не совпадет с EOF.

Одно из решений состоит в том, чтобы getint выдавала характеристику состояния файла (исчерпан или не исчерпан) в качестве результата, а значение самого числа помещала согласно указателю, переданному ей в виде аргумента. Похожая схема действует и в программе scanf, которую мы рассмотрим в параграфе 7.4. Показанный ниже цикл заполняет некоторый массив целыми числами, полученными с помощью getint.

```
int n, array[SIZE], getint (int *);
for (n = 0; n < SIZE && getint (&array[n]) != EOF; n++) ;
```

Результат каждого очередного обращения к getint посылается в array[n], и n увеличивается на единицу. Заметим, и это существенно, что функции getint передается адрес элемента array[n]. Если этого не сделать, у getint не будет способа вернуть в вызывающую программу переведенное целое число.

В предлагаемом варианте функция getint возвращает EOF по концу файла; нуль, если следующие вводимые символы не представляют собою числа; и положительное значение, если введенные символы представляют собой число.

```
#include <ctype.h>
int getch (void); void ungetch (int);
/* getint: читает следующее целое из ввода в *pn */
int getint(int *pn)
{
   int c, sign;
   while (isspace(c = getch()))
      ; /* пропуск символов-разделителей */
   if(!isdigit(c) && c != EOF && c != '+' && c != '-') {
      ungetch (c); /* не число */
      return 0;
   }
   sign =(c =='-') ? -1 : 1;
   if (c == '+' || c == '-')
      c = getch();
   for (*pn = 0; isdigit(c); c = getch())
      *pn = 10 * *pn + (c -'0');     *pn *= sign;     if (c!= EOF)
      ungetch(c);
   return c;
}
```

Везде в getint под *pn подразумевается обычная переменная типа int. Функция ungetch вместе с getch (параграф 4.3) включена в программу, чтобы обеспечить возможность отослать назад лишний прочитанный символ.

Рассмотрим небольшую программу, в которой указатель используется как аргумент функции.

```
#include <iostream>
using namespace std;
int a;
int pointers(int *b,int c);
int main(){
   cout << "Start programm" << endl;
   a=5;
   int e=8;
   cout << "a=" << a << endl;
   cout << "e=" << e << endl;
   int d = pointers(&a,e);
   cout << "a=" << a << endl;
   cout << "e=" << e << endl;
   cout << "d=" << d << endl;
   return 0;
}

int pointers(int *b,int c) {
   a++;
   (*b)++;
   int output;
   output = c + 10 ;
   return output;
}
```

Вначале объявим глобальную переменную a, которая будет доступна всем функция в текущем файле исходных кодов. Затем присвоим ей значение 5 а так же объявим локальную переменную e для функции main.

Пользовательская функция pointers в качестве входных параметров принимает указатель на тип int a так же переменную того же типа.

```
int pointers(int *b,int c);
```

В данной функции сначал производится инкремент глобальной переменной a затем инкремент объекта, на который указывает указатель b a так же возвращает c + 10.

После выполнения программы, значение a увеличится на 2, так как дважды производится операция инкремента. Первый раз – как инкремент глобальной переменной в функции, а второй раз – как инкремент объекта, на который ссылается указатель.

3.3 Указатели и массивы

В Си существует связь между указателями и массивами, и связь эта настолько тесная, что эти средства лучше рассматривать вместе. Любой доступ к элементу массива, осуществляемый операцией индексирования, может быть выполнен с помощью указателя. Вариант с указателями в общем случае работает быстрее, но разобраться в нем, особенно непосвященному, довольно трудно. Объявление int a[10];

Определяет массив a размера 10, т. е. блок из 10 последовательных объектов с именами a[0], a[1], ..., a[9].

Запись a[i] отсылает нас к i му элементу массива. Если pa есть указатель на int, т. е. объявлен как int *pa; то в результате присваивания pa = &a[0]; pa будет указывать на нулевой элемент a, иначе говоря, pa будет содержать адрес элемента a[0]. Теперь присваивание x = *pa; будет копировать содержимое a[0] в x.

Если pa указывает на некоторый элемент массива, то pa+1 по определению указывает на следующий элемент, pa+i на i й элемент после pa, а pa i на i й элемент перед pa. Таким образом, если pa указывает на a[0], то *(pa+1) есть содержимое a[1], a+i адрес a[i], а *(pa+i) содержимое a[i].

Сделанные замечания верны безотносительно к типу и размеру элементов массива a. Смысл слов "добавить 1 к указателю", как и смысл любой арифметики с указателями, состоит в том, чтобы pa+1 указывал на следующий объект, а pa+i на i й после pa.

Между индексированием и арифметикой с указателями существует очень тесная связь. По определению значение переменной или выражения типа массив есть адрес нулевого элемента массива. После присваивания pa = &a[0]; pa и a имеют одно и то же значение. Поскольку имя массива является синонимом расположения его начального элемента, присваивание pa=&a[0] можно также записать в следующем виде: pa = a;

Еще более удивительно (по крайней мере на первый взгляд) то, что a[i] можно записать как *(a+i). Вычисляя a[i], Си сразу преобразует его в *(a+i); указанные две формы записи эквивалентны. Из этого следует, что полученные в результате применения оператора & записи &a[i] и a+i также будут эквивалентными, т. е. и в том и в другом случае это адрес i го элемента после a. С другой стороны, если pa указатель, то его можно использовать с индексом, т. е. запись pa[i] эквивалентна записи *(pa+i). Короче говоря, элемент массива можно изображать как в виде указателя со смещением, так и в виде имени массива с индексом.

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие. Указатель это переменная, поэтому можно написать pa=a или pa++. Но имя массива не является переменной, и записи вроде a=pa или a++ не допускаются.

Если имя массива передается функции, то последняя получает в качестве аргумента адрес его начального элемента. Внутри вызываемой функции этот аргумент является локальной переменной, содержащей адрес. Мы можем воспользоваться отмеченным фактом и написать еще одну версию функции strlen, вычисляющей длину строки.

```
/* strlen: возвращает длину строки */ int strlen(char *s)
{
   int n;
   for (n = 0; *s != '\0'; s++)
      n++;
   return n;
}
```

Так как переменная s указатель, к ней применима операция ++; s++ не оказывает никакого влияния на строку символов функции, которая обратилась к strlen. Просто увеличивается на 1 некоторая копия указателя, находящаяся в личном пользовании функции strlen. Это значит, что все вызовы, такие как:

```
strlen("3дравствуй, мир");   /* строковая константа */
strlen(array);                /* char array[100]; */
strlen(ptr);                  /* char *ptr; */ правомерны.
```

Формальные параметры char s[]; и char *s; в определении функции эквивалентны. Мы отдаем предпочтение последнему варианту, поскольку он более явно сообщает, что s есть указатель. Если функции

в качестве аргумента передается имя массива, то она может рассматривать его так, как ей удобно либо как имя массива, либо как указатель, и поступать с ним соответственно. Она может даже использовать оба вида записи, если это покажется уместным и понятным.

Функции можно передать часть массива, для этого аргумент должен указывать на начало подмассива. Например, если a массив, то в записях

f(&a[2]) или

f(a+2)

функции f передается адрес подмассива, начинающегося с элемента a[2]. Внутри функции f описание параметров может выглядеть как f(int arr[]) {...} или f(int *arr) {...} .

Следовательно, для f тот факт, что параметр указывает на часть массива, а не на весь массив, не имеет значения.

Если есть уверенность, что элементы массива существуют, то возможно индексирование и в "обратную" сторону по отношению к нулевому элементу; выражения p[1], p[2] и т.д. не противоречат синтаксису языка и обращаются к элементам, стоящим непосредственно перед p[0]. Разумеется, нельзя "выходить" за границы массива и тем самым обращаться к несуществующим объектам.

Работа с массивом через указатель заключается в том, что объявляется указатель на тип элемента массива. Затем ему присваивается адрес первого элемента, а затем за счет инкремента адреса выполняется перебор элементов.

```
#include <iostream>
using namespace std;
void MasIncrement(int *input,int count);

int main() {
   cout << "Start program" << endl;
   int Mas[10] = {0,1,2,3,4,5,6,7,8,9};
   cout<<"Massive:"<< endl;
   for (int i=0;i<10;i++) cout<<"Mas["<<i<<"]="<<Mas[i]<<" ";
   cout<< endl;
   MasIncrement(&Mas[0],10);
   cout<<"Massive New:"<< endl;
   for (int i=0;i<10;i++) cout<<"Mas["<<i<<"]="<<Mas[i]<<" ";
   cout<< endl;
```

```
   return 0;
}

void MasIncrement(int *input,int count) {
   for (int i=0;i<count;i++) (*(input+i))++;
}
```

В функции MasIncrement() выполняется последовательный перебор элементов массива и инкремент каждого элемента.

3.4 Приведение типа указателя

Приведение указателей к типам char * или void *- гарантированное преобразование. При приведении обратно будет получен тот же указатель. При приведении к другим типам указателей и обратно обратный указатель может быть отличным от исходного.

Рассмотрим пример программы, в которой выполняется сериализация и десериализация пользовательской структуры.

```
#include <iostream>
#include <stdio.h>
using namespace std;
#pragma pack(push, 1)
struct TestUserStruct {
   int a;
   int b;
   float c;
};
#pragma pack(pop)

int main() {
   cout << "Start programm" << endl;
   TestUserStruct A;
   A.a=5;
   A.b=21;
   A.c=7.5;
   cout<<"A.a="<<A.a<<" A.b="<<A.b<<" A.c="<<A.c<<endl;
   cout<<"sizeof(A)="<<sizeof(A)<<endl;
   char Mas[sizeof(A)];
   //Сериализация сруктуры A
   char *ptr = (char*)&A;
   for (int i=0;i<sizeof(A);i++) Mas[i]=*(ptr+i);
```

```
    cout<<"Massiv:"<<endl;
    for (int i=0;i<sizeof(A);i++) printf ("Mas[%d]=%d ",i,Mas[i]);
    //Десериализация структуры B
    TestUserStruct B;
    char *e = (char*)&B;
    for (int i=0;i<sizeof(B);i++)     {
       *(e+i)=Mas[i];
    }
    cout<<endl;
    cout<<"B.a="<<B.a<<" B.b="<<B.b<<" B.c="<<B.c<<endl;
    return 0;
}
```

В данной программе создается пользовательская структура, состоящая из переменных простых типов данных. Далее производится ее сериализация в массив Mas. Затем производится десериализация данного массива во вторую копию структуры B. Если все выполнено правильно, то структуры A и B будут содержать идентичные данные.

4. Работа с файлами

Большинство компьютерных программ работают с файлами, и поэтому возникает необходимость создавать, удалять, записывать читать, открывать файлы. Что же такое файл? Файл – именованный набор байтов, который может быть сохранен на некотором накопителе. Ну, теперь ясно, что под файлом понимается некоторая последовательность байтов, которая имеет свое, уникальное имя, например файл.txt. В одной директории не могут находиться файлы с одинаковыми именами. Под именем файла понимается не только его название, но и расширение, например: file.txt и file.dat – разные файлы, хоть и имеют одинаковые названия. Существует такое понятие, как полное имя файлов – это полный адрес к директории файла с указанием имени файла, например: D:\docs\file.txt. Важно понимать эти базовые понятия, иначе сложно будет работать с файлами.

Для работы с файлами необходимо подключить заголовочный файл <fstream>. В <fstream> определены несколько классов и подключены заголовочные файлы <ifstream> – файловый ввод и <ofstream> – файловый вывод.

Файловый ввод/вывод аналогичен стандартному вводу/выводу, единственное отличие – это то, что ввод/вывод выполнятся не на эк-

ран, а в файл. Если ввод/вывод на стандартные устройства выполняется с помощью объектов cin и cout, то для организации файлового ввода/вывода достаточно создать собственные объекты, которые можно использовать аналогично операторам cin и cout.

Например, необходимо создать текстовый файл и записать в него строку Работа с файлами в С++. Для этого необходимо проделать следующие шаги:

– создать объект класса ofstream;

– связать объект класса с файлом, в который будет производиться запись;

– записать строку в файл;

– закрыть файл.

Почему необходимо создавать объект класса ofstream, а не класса ifstream? Потому, что нужно сделать запись в файл, а если бы нужно было считать данные из файла, то создавался бы объект класса ifstream.

```
// создаем объект для записи в файл
ofstream /*имя объекта*/; // объект класса ofstream
```

Назовем объект – fout, Вот что получится:

```
ofstream fout;
```

Объект необходим, чтобы можно было выполнять запись в файл. Уже объект создан, но не связан с файлом, в который нужно записать строку.

```
fout.open("cppstudio.txt"); // связываем объект с файлом
```

Через операцию точка получаем доступ к методу класса open(), в круглых скобочках которого указываем имя файла. Указанный файл будет создан в текущей директории с программой. Если файл с таким именем существует, то существующий файл будет заменен новым. Итак, файл открыт, осталось записать в него нужную строку. Делается это так:

```
fout << "Работа с файлами в С++"; // запись строки в файл
```

Используя операцию передачи в поток совместно с объектом fout строка Работа с файлами в С++ записывается в файл. Так как больше нет необходимости изменять содержимое файла, его нужно закрыть, то есть отделить объект от файла.

```
fout.close(); // закрываем файл
```

Итог – создан файл со строкой Работа с файлами в С++.

Шаги 1 и 2 можно объединить, то есть в одной строке создать объект и связать его с файлом. Делается это так:

ofstream fout("cppstudio.txt"); // создаем объект класса ofstream и связываем его с файлом cppstudio.txt

Объединим весь код и получим следующую программу.

// file.cpp: определяет точку входа для консольного приложения.

```
#include "stdafx.h"
#include <fstream>
using namespace std;

int main(int argc, char* argv[])
{
   ofstream fout("cppstudio.txt"); // создаем объект класса ofstream для записи и //связываем его с файлом cppstudio.txt
   fout << "Работа с файлами в C++"; // запись строки в файл
   fout.close(); // закрываем файл
   system("pause");
   return 0;
}
```

Осталось проверить правильность работы программы, а для этого открываем файл cppstudio.txt и смотрим его содержимое, должно быть – Работа с файлами в C++.

Для того чтобы прочитать файл понадобится выполнить те же шаги, что и при записи в файл с небольшими изменениями:

– создать объект класса ifstream и связать его с файлом, из которого будет производиться считывание;

– прочитать файл;

– закрыть файл.

```
// file_read.cpp: определяет точку входа для консольного приложения.

#include "stdafx.h"
#include <fstream>
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
   setlocale(LC_ALL, "rus"); // корректное отображение Кириллицы
```

```
    char buff[50]; // буфер промежуточного хранения считываемого из файла текста
    ifstream fin("cppstudio.txt"); // открыли файл для чтения

    fin >> buff; // считали первое слово из файла
    cout << buff << endl; // напечатали это слово

    fin.getline(buff, 50); // считали строку из файла
    fin.close(); // закрываем файл
    cout << buff << endl; // напечатали эту строку

    system("pause");
    return 0;
}
```

В программе показаны два способа чтения из файла, первый – используя операцию передачи в поток, второй – используя функцию getline(). В первом случае считывается только первое слово, а во втором случае считывается строка, длинной 50 символов. Но так как в файле осталось меньше 50 символов, то считываются символы включительно до последнего. Обратите внимание на то, что считывание во второй раз (строка 17) продолжилось, после первого слова, а не с начала, так как первое слово было прочитано в строке 14. Результат работы программы показан на рисунке 4.1.

```
Работа
 с файлами в C++
Для продолжения нажмите любую клавишу . . .
```

Рисунок 4.1 – Результат работы программы

Программа сработала правильно, но не всегда так бывает, даже в том случае, если с кодом все в порядке. Например, в программу передано имя несуществующего файла или в имени допущена ошибка. Что тогда? В этом случае ничего не произойдет вообще. Файл не будет найден, а значит и прочитать его не возможно. Поэтому компилятор проигнорирует строки, где выполняется работа с файлом. В результате корректно завершится работа программы, но ничего на экране показано не будет. Казалось бы, это вполне нормальная реакция на такую ситуацию. Но простому пользователю не будет понятно, в чем дело и почему на экране не появилась строка из файла. Так вот, чтобы все было предельно понятно в C++ предусмотрена такая функция –

is_open(), которая возвращает целые значения: 1 – если файл был успешно открыт, 0 – если файл открыт не был. Доработаем программу с открытием файла, таким образом, что если файл не открыт выводилось соответствующее сообщение.

```
// file_read.cpp: определяет точку входа для консольного приложения.

#include "stdafx.h"
#include <fstream>
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
   setlocale(LC_ALL, "rus"); // корректное отображение Кириллицы
   char buff[50]; // буфер промежуточного хранения считываемого из файла текста
   ifstream fin("cppstudio.doc"); // (ВВЕЛИ НЕ КОРРЕКТНОЕ ИМЯ ФАЙЛА)

   if (!fin.is_open()) // если файл не открыт
      cout << "Файл не может быть открыт!\n"; // сообщить об этом
   else
   {
   fin >> buff; // считали первое слово из файла
   cout << buff << endl; // напечатали это слово

   fin.getline(buff, 50); // считали строку из файла
   fin.close(); // закрываем файл
   cout << buff << endl; // напечатали эту строку
   }
   system("pause");
   return 0;
}
```

Результат работы программы показан на рисунке 4.2.

```
Файл не может быть открыт!
Для продолжения нажмите любую клавишу . . .
```

Рисунок 4.2 – Работа с файлами в C++

Как видно из рисунка 4.2 программа сообщила о невозможности открыть файл. Поэтому, если программа работает с файлами, рекомендуется использовать эту функцию, is_open(), даже, если уверены, что файл существует.

Режимы открытия файлов устанавливают характер использования файлов. Для установки режима в классе ios_base предусмотрены константы, которые определяют режим открытия файлов (см. таблицу 4.1).

Таблица 4.1 – Режимы открытия файлов

Режим	**Описание**
ios_base::in	открыть файл для чтения
ios_base::out	открыть файл для записи
ios_base::ate	при открытии переместить указатель в конец файла
ios_base::app	открыть файл для записи в конец файла
ios_base::trunc	удалить содержимое файла, если он существует
ios_base::binary	открытие файла в двоичном режиме

Режимы открытия файлов можно устанавливать непосредственно при создании объекта или при вызове функции open().

```
ofstream fout("cppstudio.txt", ios_base::app); // открываем файл для добавления
//информации к концу файла
fout.open("cppstudio.txt", ios_base::app); // открываем файл для добавления информации
 //к концу файла
```

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции или |, например: ios_base::out | ios_base::trunc – открытие файла для записи, предварительно очистив его.

Объекты класса ofstream, при связке с файлами по умолчанию содержат режимы открытия файлов ios_base::out | ios_base::trunc. То есть файл будет создан, если не существует. Если же файл существует, то его содержимое будет удалено, а сам файл будет готов к записи. Объекты класса ifstream связываясь с файлом, имеют по умолчанию режим открытия файла ios_base::in – файл открыт только для чтения. Режим открытия файла еще называют – флаг, для удобочитаемости в дальнейшем будем использовать именно этот термин. В таб-

лице 1 перечислены далеко не все флаги, но для начала этих должно хватить.

Обратите внимание на то, что флаги ate и app по описанию очень похожи, они оба перемещают указатель в конец файла, но флаг app позволяет производить запись, только в конец файла, а флаг ate просто переставляет флаг в конец файла и не ограничивает места записи.

Разработаем программу, которая, используя операцию sizeof(), будет вычислять характеристики основных типов данных в C++ и записывать их в файл. Характеристики:

1) число байт, отводимое под тип данных;

2) максимальное значение, которое может хранить определенный тип данных.

Запись в файл должна выполняться в таком формате:

```
/*  data type          byte      max value
bool                   1         255.00
char                   1         255.00
short int              2         32767.00
unsigned short int     2         65535.00
int                    4         2147483647.00
unsigned int           4         4294967295.00
long int               4         2147483647.00
unsigned long int      4         4294967295.00
float                  4         2147483647.00
long float             8         92233720368547758O0.00
double                 8         92233720368547758O0.00  */
```

Необходимо открыть файл в режиме записи, с предварительным усечением текущей информации файла (строка 14). Как только файл создан и успешно открыт (строки 16 – 20), вместо оператора cout, в строке 22 используем объект fout. таким образом, вместо экрана информация о типах данных запишется в файл.

```
// write_file.cpp: определяет точку входа для консольного приложения.

#include "stdafx.h"
#include <iostream>
#include <fstream> // работа с файлами
#include <iomanip> // манипуляторы ввода/вывода
using namespace std;

int main(int argc, char* argv[])
```

```
{
   setlocale(LC_ALL, "rus");

   // связываем объект с файлом, при этом файл открываем в режиме записи, предварительно удаляя все данные из него
   ofstream fout("data_types.txt", ios_base::out | ios_base::trunc);

   if (!fout.is_open()) // если файл небыл открыт
   {
    cout << "Файл не может быть открыт или создан\n"; // напечатать соответствующее сообщение
    return 1; // выполнить выход из программы
   }

      fout << "     data type      "   << "byte"                 << "      "    << "max value  " << endl // заголовки столбцов
         << "bool              =  " << sizeof(bool)              << "         " << fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных bool*/ << (pow(2,sizeof(bool) * 8.0) - 1)          << endl
       << "char              =  " << sizeof(char)              << "         " << fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных char*/ << (pow(2,sizeof(char) * 8.0) - 1)          << endl
       << "short int         =  " << sizeof(short int)         << "         " << fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных short int*/ << (pow(2,sizeof(short int) * 8.0 - 1) - 1)     << endl
         << "unsigned short int =  " << sizeof(unsigned short int)  << "  " << fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных unsigned short int*/          << (pow(2,sizeof(unsigned short int) * 8.0) - 1) << endl
         << "int               =  " << sizeof(int)               << "         " << fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных int*/ << (pow(2,sizeof(int) * 8.0 - 1) - 1)          << endl
         << "unsigned int      =  " << sizeof(unsigned int)      << "         " << fixed << setprecision(2)
```

```
/*вычисляем максимальное значение для типа данных unsigned int*/
<< (pow(2,sizeof(unsigned int) * 8.0) - 1)      << endl
        << "long int          =  " << sizeof(long int)          << "        " <<
fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных long int*/
<< (pow(2,sizeof(long int) * 8.0 - 1) - 1)      << endl
        << "unsigned long int  =  " << sizeof(unsigned long int)    << "
" << fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных undigned long
int*/           << (pow(2,sizeof(unsigned long int) * 8.0) - 1)  << endl
        << "float             =  " << sizeof(float)             << "        " << fixed
<< setprecision(2)
/*вычисляем максимальное значение для типа данных float*/
      << (pow(2,sizeof(float) * 8.0 - 1) - 1)       << endl
        << "long float        =  " << sizeof(long float)        << "        " <<
fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных long float*/
<< (pow(2,sizeof(long float) * 8.0 - 1) - 1)    << endl
        << "double            =  " << sizeof(double)            << "        " <<
fixed << setprecision(2)
/*вычисляем максимальное значение для типа данных double*/
<< (pow(2,sizeof(double) * 8.0 - 1) - 1)        << endl;
      fout.close(); // программа больше не использует файл, поэтому его
нужно закрыть
   cout << "Данные успешно записаны в файл data_types.txt\n";
   system("pause");
   return 0;
}
```

Нельзя не заметить, что изменения в программе минимальны, а все благодаря тому, что стандартный ввод/вывод и файловый ввод/вывод используются абсолютно аналогично. В конце программы, в строке 45 мы явно закрыли файл, хотя это и не обязательно, но считается хорошим тоном программирования. Стоит отметить, что все функции и манипуляторы, используемые для форматирования стандартного ввода/вывода, актуальны и для файлового ввода/вывода. Поэтому не возникло никаких ошибок, когда оператор cout был заменен объектом fout.

5. Работа с классами

5.1 Общие подходы

Объектно-ориентированная технология (парадигма) программирования наиболее распространена и востребована в настоящее время. При объектно-ориентированном подходе к программированию программа представляет собой совокупность взаимодействующих между собой данных – объектов. Функциональную возможность и структуру объектов задают классы – типы данных, определенные пользователем.

В соответствии с концепцией фон-Неймана – основателя теоретической концепции компьютерной техники, процессор обрабатывает данные, выполняя инструкции (команды), которые находятся в той же оперативной памяти, что и данные.

Таким образом, можно выделить две основные сущности процесса обработки информации: код, как совокупность инструкций, и данные. Все программы в соответствии с выбранной технологией программирования концептуально организованы вокруг своего кода или вокруг своих данных. Рассмотрим основные на сегодняшний день парадигмы программирования:

1. Процессно-ориентированная парадигма, при которой программа представляет собой ряд последовательно выполняемых операций – модель фон-Неймана. При этом код воздействует на данные. Языки, реализующие эту парадигму, называются процедурными или императивными. Такими языками являются, например, C, Pascal и др.

2. Объектно-ориентированная парадигма, при которой программа рассматривается как совокупность фрагментов кода, обрабатывающих отдельные совокупности данных – объекты. Эти объекты взаимодействуют друг с другом посредством так называемых интерфейсов. При этом данные управляют доступом к коду.

Центральной идеей ООП является реализация понятия "абстракция". Смысл абстракции заключается в том, что сущность произвольной сложности можно рассматривать, а также производить определенные действия над ней, как над единым целым, не вдаваясь в детали внутреннего построения и функционирования.

При создании программного комплекса необходимо разработать определенные абстракции.

Пример: Задача составления расписания занятий.

Необходимые абстракции: студент, курс лекций, преподаватель, аудитория.

Операции:

− Определить студента в группу − Назначить аудиторию для группы

− . . . . . .

Во всех объектно-ориентированных языках программирования реализованы следующие основные механизмы (постулаты) ООП:

Инкапсуляция.

Наследование.

Полиморфизм.

Инкапсуляция – механизм, связывающий вместе код и данные, которыми он манипулирует, и одновременно защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому. Доступ к коду и данным жестко контролируется интерфейсом. Основой инкапсуляции при ООП является класс. Механизма инкапсуляции позволяет оставлять скрытыми от пользователя некоторые детали реализации класса (т. е. инкапсулировать их в классе), что упрощает работу с объектами этого класса.

Наследование – механизм, с помощью которого один объект (производного класса) приобретает свойства другого объекта (родительского, базового класса). При использовании наследования новый объект не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста. Наследование позволяет какому-либо объекту наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Наследование есть очень важное понятие, поддерживающее концепцию иерархической классификации.

Полиморфизм – механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

Общая концепция полиморфизма: один интерфейс – много методов. Выбор конкретного действия (метода) применительно к конкретной ситуации возлагается на компилятор. Программисту же достаточно запомнить и применить один интерфейс, вместо нескольких, что также упрощает работу.

Различаются статический (реализуется на этапе компиляции с помощью перегрузки функций и операций), динамический (реализуется во время выполнения программы с помощью механизма вирту-

альных функций) и параметрический (реализуется на этапе компиляции с использованием механизма шаблонов) полиморфизм.

5.2 Классы и объекты C++

Центральным понятием ООП является класс. Класс используется для описания типа, на основе которого создаются объекты (переменные типа класс).

Класс, как и любой тип данных, характеризуется множеством значений, которые могут принимать объекты класса, и множеством функций, задающих операции над объектами.

Синтаксис описания класса

class Имя_класса { определение_членов_класса };

Члены класса можно разделить на информационные и функции (методы). Информационные члены описывают внутреннюю структуру информации, хранящейся в объекте, который создается на основе класса. Методы класса описывают алгоритмы обработки этой информации.

Данные, хранящиеся в информационных членах, описывают состояние объекта, созданного на основе класса. Состояние объекта изменяется на основе изменения хранящихся данных с помощью методов класса. Алгоритмы, заложенные в реализации методов класса, определяют поведение объекта, то есть реагирование объекта на поступающие внешние воздействия в виде входных данных.

Принцип инкапсуляции обеспечивается вводом в класс областей доступа:

– private (закрытый, доступный только собственным методам)

– public (открытый, доступный любым функциям)

– protected (защищенный, доступный только собственным методам и методам производных классов)

Члены класса, находящиеся в закрытой области (private), недоступны для использования со стороны внешнего кода. Напротив, члены класса, находящиеся в открытой секции (public), доступны для использования со стороны внешнего кода. При описании класса каждый член класса помещается в одну из перечисленных выше областей доступа следующим образом:

```
class Имя_класса {
private:
определение_закрытых_членов_класса
    public:
```

```
        определение_открытых_членов_класса
    protected:
        определение_защищенных_членов_класса
 …
};
```

Порядок следования областей доступа и их количество в классе – произвольны.

Служебное слово, определяющее первую область доступа, может отсутствовать. По умолчанию эта область считается private.

В закрытую (private) область обычно помещаются информационные члены, а в открытую (public) область – методы класса, реализующие интерфейс объектов класса с внешней средой. Если какой-либо метод имеет вспомогательное значение для других методов класса, являясь подпрограммой для них, то его также следует поместить в закрытую область. Это обеспечивает логическую целостность информации.

После описания класса его имя можно использовать для описания объектов этого типа. Доступ к информационным членам и методам объекта, описанным в открытой секции, осуществляется через объект или ссылку на объект с помощью операции выбора члена класса ‘.’.

Если работа с объектом выполняется с помощью указателя на объект, то доступ к соответствующим членам класса осуществляется на основе указателя на член класса‘->’:

```
class X
{
public:
        char c;
int f(){…}
};

int main () {
        X x1;
        X & x2 = x1;
        X * p = & x1; int i, j, k; x1.c = ‘*’;
        i = x1.f(); x1.c = '+';
        j = x2.f(); x1.c = ‘#’;
        k = p -> f();
```

```
    …
}
```

Объекты класса можно определять совместно с описанием класса:

class Y {…} y1, y2;

5.3 Объявление и определение методов класса. Спецификатор inline

Каждый метод класса, должен быть определен в программе. Определить метод класса можно либо непосредственно в классе (если тело метода не слишком сложно и громоздко), либо вынести определение вне класса, а в классе только объявить соответствующий метод, указав его прототип.

При определении метода класса вне класса для указания области видимости соответствующего имени метода используется операции ‘::’:

Пример:

```
class x {       int ia1;    public:
    x(){ia1 = 0;}       int func1();
};

int x::func1(){ … return ia1; }
```

Это позволяет повысить наглядность текста, особенно, в случае значительного объема кода в методах. При определении метода вне класса с использованием операции ‘::’ прототипы объявления и определения функции должны совпадать.

Метод класса и любую функцию, не связанную ни с каким классом, можно определить со спецификатором inline:

inline int func1();

Такие функции называются встроенными.

Спецификатор inline указывает компилятору, что необходимо по возможности генерировать в точке вызова код функции, а не команды вызова функции, находящейся в отдельном месте кода модуля. Это позволяет уменьшить время выполнения программы за счет отсутствия команд вызова функции и возврата из функции, которые кроме

передачи управления выполняют действия соответственно по сохранению и восстановлению контекста (содержимого основных регистров процессора). При этом размер модуля оказывается увеличенным по сравнению с программой без спецификаторов inline. Следует отметить, что спецификатор inline является рекомендацией компилятору. Данный спецификатор неприменим для функций со сложной логикой. В случае невозможности использования спецификатора для конкретной функции компилятор выдает предупреждающее сообщение и обрабатывает функции стандартным способом.

По определению методы класса, определенные непосредственно в классе, являются inline-функциям.

5.4 Конструкторы и деструкторы

Конструкторы и деструкторы являются специальными методами класса. Конструкторы вызываются при создании объектов класса и отведении памяти под них. Деструкторы вызываются при уничтожении объектов и освобождении отведенной для них памяти.

В большинстве случаев конструкторы и деструкторы вызываются автоматически (неявно) соответственно при описании объекта (в момент отведения памяти под него) и при уничтожении объекта. Конструктор (как и деструктор) может вызываться и явно, например, при создании объекта в динамической области памяти с помощью операции new.

Так как конструкторы и деструкторы неявно входят в интерфейс объекта, их следует располагать в открытой области класса.

Отличия и особенности описания конструктора от обычной функции:

- Имя конструктора совпадает с именем класса
- При описании конструктора не указывается тип возвращаемого значения

Следует отметить, что и обычная процедура может не возвращать значения, а только перерабатывать имеющиеся данные. В этом случае при описании соответствующей функции указывается специальный тип возвращаемого значения void.

В описании конструктора тип возвращаемого значения не указывается не потому, что возвращаемого значения нет. Оно как раз есть. Ведь результатом работы конструктора в соответствии с его названием является созданный объект того типа, который описывается данным классом. Страуструп отмечал, что конструктор – это то, что область памяти превращает в объект.

Конструкторы можно классифицировать разными способами:

1) по наличию параметров:

- без параметров;
- с параметрами;

2) по количеству и типу параметров:

- конструктор умолчания;
- конструктор преобразования;
- конструктор копирования;
- конструктор с двумя и более параметрами.

Набор и типы параметров зависят от того, на основе каких данных создается объект.

В классе может быть несколько конструкторов. В соответствии с правилами языка C++ все они имеют одно имя, совпадающее с именем класса, что является одним из проявлений статического полиморфизма. Компилятор выбирает тот конструктор, который в зависимости от ситуации, в которой происходит создание объекта, удовлетворяет ей по количеству и типам параметров. Естественным ограничением является то, что в классе не может быть двух конструкторов с одинаковым набором параметров.

Деструкторы применяются для корректного уничтожения объектов. Часто процесс уничтожения объектов включает в себя действия по освобождению выделенной для них по операциям new памяти.

Имя деструктора: ~имя_класса

У деструкторов нет параметров и возвращаемого значения.

В отличие от конструкторов, деструктор в классе может быть только один.

Пример: Описание класса.

```
class box{
    int len, wid, hei;
public:
    box(int l, int w, int h){
        len = l; wid = w; hei = h;
    }
    box(int s){
        len = wid = hei = s; 15
     }
    box(){
        len = 2;
        wid = hei = 1;
```

```
    }
    int volume(){
        return len * wid * hei;
    }
};
```

5.5 Конструктор умолчания

Конструктор без параметров называется конструктором умолчания.

Если для создания объекта не требуется каких-либо параметров, то используется конструктор умолчания. При описании таких объектов после имени класса указывается только идентификатор переменной:

class X{ … }; X x1;

Замечание: роль конструктора умолчания может играть конструктор, у которого все параметры имеют априорные значения, например:

box (int l = 24, int w = 12, int h = 6);

Конструктор преобразования и конструкторы с двумя и более параметрами

Если для создания объекта необходимы параметры, то они указываются в круглых скобках после идентификатора переменной:

box b2(1,2,3); box b3(5);

Указываемые параметры являются параметрами конструктора класса. Если у конструктора имеется ровно один входной параметр, который не представляет собой ссылку на свой собственный класс, то соответствующий конструктор называется конструктором преобразования. Этот конструктор называется так в связи с тем, что в результате его работы на основе объекта одного типа создается объект другого типа (типа описываемого класса).

Если уже описан класс T и описывается новый класс X, то его конструкторы преобразования могут иметь любой из следующих прототипов:

X(T);
X(T&);
X(const T&);

Последний прототип служит для защиты от изменения передаваемого фактического параметра в теле конструктора, так как при получении ссылки на фактический параметр используется собственно передаваемый объект, а не его локальная копия.

Vector v1(10);
Matrix m1(10,15);

Используется в первом случае один параметр, а во втором случае – два параметра. Таким образом, в первом случае объект создается с помощью конструктора преобразования, а во втором случае, с формальной точки зрения, с помощью конструктора с двумя параметрами, хотя в обоих случаях фактически выполняется одна и та же процедура: создание объекта на основе заданных числовых параметров.

Как уже было отмечено, если у параметра конструктора преобразования имеется априорное значение, и при описании объекта явно не задается фактический параметр, этот конструктор играет роль конструктора умолчания.

Пример:

```
class X {
int x1;
public:
X(int px1 = 0}
};
```

Для такого класса будут верны следующие объявления объектов:

```
int main(){
… X x1, x2(1); …
}
```

5.6 Конструктор копирования

При создании объекта его информационные члены могут быть проинициализированы значениями полей другого объекта этого же типа, т. е. объект создается как копия другого объекта.

Для такого создания объекта используется конструктор копирования.

Инициализация может быть выполнена аналогично инициализации переменных встроенных типов с использованием операции присваивания совместно с объявлением объекта:

```
box b5(2,4,6); // создание объекта типа box с
               // использованием числовых данных
box b6 = b5;   // создание объекта b6 – копии объекта b5
```

Если инициализация производится объектом такого же типа, то объект-инициализатор также может быть указан в круглых скобках после идентификатора создаваемого объекта:

```
box b7(b5);
```

Если класс не предусматривает создания внутренних динамических структур, например, массивов, создаваемых с использованием операции new, то в конструкторе копирования достаточно предусмотреть поверхностное копирование, то есть почленное копирование информационных членов класса.

Конструктор копирования, осуществляющий поверхностное копирование, можно явно не описывать, он сгенерируется автоматически.

Если же в классе предусмотрено создание внутренних динамических структур, использование только поверхностного копирования будет ошибочным, так как информационные члены-указатели, находящиеся в разных объектах, будут иметь одинаковые значения и указывать на одну и ту же размещенную в динамической памяти структуру. Автоматически сгенерированный конструктора копирования в данном классе не позволит корректно создавать объекты такого типа на основе других объектов этого же типа.

В подобных случаях необходимо глубокое копирование, осуществляющее не только копирование информационных членов объекта, но и самих динамических структур. При этом, естественно, информационные члены-указатели в создаваемом объекте должны не механически копироваться из объекта-инициализатора, а указывать на вновь созданные динамические структуры нового объекта.

Пример: Для класса stack конструктор копирования может быть определен следующим образом:

```
class stack {
        char*c1;
        int top, size;
public:
        stack(int n = 10){ size = n; top = 0; c1 = new char[size];
}
stack(stack & s1);
. . .
};
stack::stack(stack & s1){
        size = s1.size;
        top = s1.top;
        c1 = new char[size];
for (int i = 0; i < size; i++)
        c1[i] = s1.c1[i];
}
```

Замечания по работе конструктора копирования:

– Входной параметр является внешним объектом по отношению к создаваемому объекту. Тем не менее, имеется возможность прямого обращения к закрытым членам этого внешнего объекта. Это возможно только потому, что входной параметр имеет тип, совпадающий с типом создаваемого в результате работы конструктора копирования объекта. Если бы на вход конструктора поступал бы объект другого типа (например, в конструкторе преобразования класса vector входным параметром был бы объект, созданный на основе класса matrix), то для доступа к закрытым членам объекта-параметра необходимо было бы применять специальные средства. Это связано с тем, что единицей защиты является не объект, а тип, то есть методы объекта могут обращаться к закрытым членам не только данного объекта, но и к закрытым членам любого объекта данного типа.

– В момент описания конструктора копирования класс, как тип данных, еще не описан до конца. Тем не менее, идентификатор класса уже используется в качестве полноценного типа данных при описании входного параметра конструктора копирования. Такая технология схожа с описанием рекурсивной функции, когда тело описываемой функции содержит вызов этой же функции.

В отличие от конструктора преобразования, входной параметр конструктора копирования имеет тип, описываемый данным классом. Таким образом, если описывается класс X, то его конструктор копирования может иметь один из следующих прототипов:

X(X&);

X(const X&);

Объект, создаваемый с использованием конструктора копирования, может инициализироваться не только именованными объектами, но и временно созданными объектами.

Пример:

```
box* b4 = new box(2,3,5);     // Явный запуск конструктора
                              // с тремя параметрами. Адрес
                              // динамически созданного
                              // объекта типа box
                              // присваивается переменной
                               // b4.
box b5 = * b4;                      // Разыменование указателя на объект,
                              // т.е. получение доступа к
                              // информации, хранящейся в нем, и
```

```
                    //  использование ее для инициализации
                    // создаваемого объекта.
box b6 = box(4,7,1);          // Создание временного объекта и
                    // инициализация именованного
                    // объекта.
```

5.7 Автоматическая генерация конструкторов и деструкторов

Автоматически могут генерироваться только конструкторы умолчания, конструкторы копирования и деструкторы. Если в классе явно не описано ни одного конструктора, то автоматически генерируется конструктор умолчания с пустым телом.

Если в классе явно описан хотя бы один конструктор, например, конструктор копирования, то конструктор умолчания не будет автоматически генерироваться, даже, если он необходим в соответствии с постановкой задачи.

В случае отсутствия в классе явно описанного конструктора копирования он всегда генерируется автоматически и обеспечивает поверхностное копирование.

Если в классе не описан деструктор, то всегда автоматически генерируется деструктор, который не производит никаких действий.

Таким образом, даже если в классе не описаны конструкторы и деструктор, они все равно неявно присутствуют в нем.

5.8 Статические методы и данные.

Некоторые члены класса могут быть объявлены с модификатором класса памяти static. Это статические члены класса. Статические члены-данные класса являются общими для всех объектов данного класса. Изменив значение статического члена класса в одном объекте, мы получим изменившееся значение во всех других объектах класса.

Статические члены – данные класса можно использовать для подсчета количества созданных объектов класса или существующих в данный момент объектов класса.

Методы класса также могут быть объявлены статическими. Статические методы класса не получают указатель this, соответственно, эти функции не могут обращаться к нестатическим членам класса. К статическим членам класса статические методы класса обращаются посредством операции точка или ->. Статический метод класса не может быть виртуальным. К статическим методам класса можно обращаться, даже если не создано ни одного объекта данного класса, нуж-

но только использовать полное имя члена класса. Если метод func1() является статичческим методом класса A, то ее можно вызвать: A::func1();

```
#include <iostream>
using namespace std;

class gamma
{
private:
	static int total;  //всего объектов класса
	//(только объявление)
	int id;  //ID текущего объекта
public:
	gamma()  //конструктор без аргументов
	{
		total++;  //добавить объект
		id = total;  //id равен текущему значению total
	}
	~gamma()  //деструктор
	{
		total;
		cout << "Удаление ID " << id << endl;
	}
	static void showtotal()  // статическая функция
	{
		cout << "Всего: " << total << endl;
	}
	void showid()  // Нестатическая функция
	{
		cout << "ID: " << id << endl;
	}
};
//-----------------------------------------------
int gamma::total = 0;  // определение total

int main()
{
	gamma g1;
	gamma::showtotal();
```

```
        //без static обращение к методу только через объект
        gamma g2, g3;
        gamma::showtotal();
        g1.showid();
        g2.showid();
        g3.showid();
        cout << "конец программы\n";
return 0;
}
```

5.9 Методы const, не изменяющие объекты класса.

Const-метод – это обычный метод, который, помимо своих прямых обязательств, дает гарантию того, что он не изменит атрибуты объекта. Любая попытка нарушить эту гарантию будет пресекаться компилятором. Определяется такой метод наличием ключевого слова const в конце сигнатуры метода. Рассмотрим на примере.

Есть некоторый класс Foo, метод doSomething() которого присваивает переменной члену _x значение 5.Методы const, не изменяющие объекты класса

```
class Foo {
public:
   Foo(): _x(0) {}
   void doSomething();

private:
   int _x;
};

void Foo::doSomething()
{
   _x = 5;
}
```

В данном классе метод doSomething изменяет знаение переменной X. В таком случае класс отработает и никаких проблем не возникнет. А в случае если данный метод будет объявлен с можификатором const, дело обстоит по иному:

```
class Foo {
public:
   Foo(): _x(0) {}
   void doSomething()const;

private:
   int _x;
};

void Foo::doSomething()const {
   _x = 5;     // для const-метода изменение атрибутов запрещено
}
```

То компилятор ругнется на строку, и напишет (в случае gcc):

```
test.cpp: In member function ‘void Foo::doSomething() const’:
test.cpp:15:10: error: assignment of member ‘Foo::_x’ in read-only object
```

5.10 Правила наследования

Наследование является одним из трех основных механизмов ООП. В результате использования механизма наследования (рисунок 5.1) осуществляется формирование иерархических связей между описываемыми типами. Тип-наследник уточняет базовый тип.

Пример:

```
struct A {int x,y;};
struct B: A {int z;};
A a1;
B b1;
b1.x =1;
b1.y =2;
b1.z =3;
a1 = b1;
```

Объект типа B наследует свойства объекта типа A. Таким образом, объект типа-наследника содержит внутри себя члены базового типа.

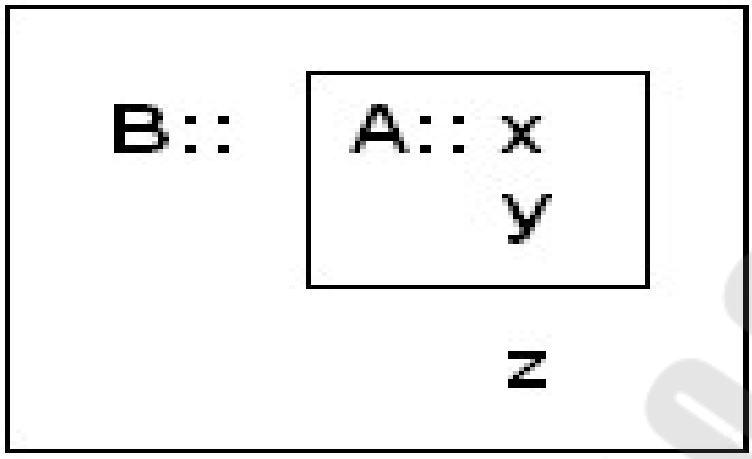

Рисунок 5.1 – Принцип наследования

Наследование – процесс создания новых классов, называемых наследниками или производными, из уже существующих, базовых классов. Производный класс получает все возможности базового класса, но может быть усовершенствован. Основная форма наследования:

class имя_производного_класса : режим_доступа
имя_базового_класса

При наследовании наследуются не только информационные члены, но и методы. Не наследуются:

- Конструкторы.
- Деструктор.
- Операция присваивания.

5.11 Управление уровнем доступа к элементам класса

Наследование свойств и поведения могут контролироваться с помощью квалификаторов доступа, задаваемых при наследовании: public, protected, private. Названия квалификаторов доступа совпадают с названиями квалификаторов доступа к методам и членам класса. Квалификаторы доступа ограничивают видимость полностью или частично для полностью или частично открытых членов-данных и методов. Закрытые члены-данных и методы всегда остаются закрытыми. При наследовании можно уменьшить видимость, но не расширить.

Кратко вид доступа в типе-наследнике для членов базового типа можно представить таблицей.

Таблица 5.1 – Вид доступа

Область доступа базового типа	**Квалификатор доступа**	**public**	**protected**	**private**
public		public	protected	private
protected		protected	protected	private
private		private	private	private

Данная таблица показывает вид доступа для членов-данных и методов в типе наследнике для типа наследника следующего уровня. Закрытый вид доступа в типе-наследнике для закрытых членов базового типа имеет особый статус.

Если не указан тип наследования, то тип наследования по умолчанию определяется описанием типа наследника. Если тип-наследник описывается классом, то тип наследования – закрытый (private), если же это структура, то наследование по умолчанию будет открытым (public). Пример:

```
 struct A { int x; };
class C: A {};
int main(){
        C c;
        c.x = 1; // ошибка: в классе C из-за закрытого по
                 // умолчанию наследования поле x
                 // становится закрытым.
return 0;
}
```

Если тип-наследник описывается структурой, то наследование по умолчанию становится открытым. Пример:

```
class A {
        public:
                int x;
        private:
                int y;
};
struct C: A {};

int main(){
        C c;
```

```
    c.x = 1;     // ошибки нет, так как наследование – открытое.
  return 0;
}
```

При необходимости открытого наследования членов базового типа, если тип-наследник описывается с использованием класса, следует явно указывать квалификатор public:

```
class C: public A { int z; };
```

Защищенный вид доступа (protected) означает, что члены базового типа в типе-наследнике доступны только для методов своего (базового) типа, а также для методов производного типа. Во всех остальных случаях они ведут себя так же, как члены с закрытым видом доступа (private). Пример:

```
struct A {    int x,y;
    };
class B: protected A {
    int z;
public:
    void putx(int ap) { x = ap;
    }
};

int main(){
        B b1;
        b1.putx(1);
    . . .
    return 0;
}
```

Ограничение видимости при наследовании ограничивает манипуляции с членами базового типа только в объектах типа-наследника и его потомках. Поэтому при преобразовании указателя типа-наследника к указателю на объекты базового типа работа с объектом осуществляется в соответствии с правилами видимости для базового класса.

Пусть указатель на объект типа-наследника при защищенном наследовании преобразован к указателю на объекты базового типа. Тогда работа с объектом типа-наследника с использованием указателя на объекты базового типа происходит в соответствии с правами доступа для базового типа (как уже было указано, через такой указатель виден не весь объект типа-наследника, а только его часть, соответствующая базовому типу):

```
struct A {int x; int y;};
struct B: A {int z;};

class C: protected A {int z;};
int main(){
        A a;
        A* pa;
        B b;
        C c;
        C* pc = &c;
        b.x = 1;
        pc -> z;       // ошибка: доступ к закрытому полю
        b.y = 2;
        pc -> x;       // ошибка: доступ к закрытому полю
        b.z = 3;
        pa = (A*)pc;        // см. примечание далее.
        a = b;
        pa -> x=4;  // правильно: поле A::x - открытое
return 0;
}
```

Так как в данном примере наследование – защищенное, то при присвоении указателя производного типа указателю базового типа требуется явное преобразование (pa=(A*)pc;). При открытом наследовании возможно простое присвоение указателей (pa=pc). Это связано с тем, что указатель кроме адреса содержит информацию об объекте. При защищенном наследовании изменяется не только состав членов класса, но и права доступа.

Закрытые члены базового класса недоступны напрямую с использованием дополнительных методов класса-наследника (при любом способе наследования). Работа внутри класса-наследника с такими получаемыми закрытыми членами базового класса возможна только с использованием открытых и защищенных методов базового класса.

Закрытые и защищенные получаемые методы недоступны для манипулирования с объектом вне класса. Они могут использоваться как подпрограммы другими методами класса.

При закрытом наследовании открытые и защищенные члены базового класса (любые) доступны только внутри производного класса и недоступны извне (через объекты производного класса), как и его собственные закрытые члены.

Таким образом, приведенная таблица показывает вид доступа для членов в типе наследнике для типа наследника следующего уровня. Но, для текущего типа-наследника доступность зависит от вида доступа в базовом типе. Пример:

```
class X1{
        int ix1;
public:
        int f1(){ . . . }
        . . .
};
class Y1: protected X1 {
. . .
};
class Z1: public Y1 {
. . .
};
class X2{   protected:      int ix2;   public:
     int f2(){ . . . }
     . . . };
class Y2: X2 {. . .};
class Z2: public Y2 {. . .};
```

В классе Y1 переменная ix1 недоступна непосредственно, так как она в базовом классе X1 находится в закрытой области. Однако она может быть использована в функции f1(). В то же время функция f1() не может использоваться в качестве внешнего метода по отношению к объекту, созданному на основе класса Y1, так как она находится в защищенной области класса Y1. Это же остается справедливым и для класса Z1. В классе Y2 переменная ix2, в отличие от переменной ix1 в классе Y1, доступна непосредственно. В классе Z2 переменная ix2 становится недоступной для непосредственного использования, так же, как и переменная ix1 в классе Z1. Другие отличия классов Z1 и Z2: в отличие от функции f1() в классе Z1, функция f2() в классе Z2 доступна в классе Z2 в качестве внутренней подпрограммы, только для функций, унаследованных из класса Y2, так как в классе Y2 она находится в закрытой области.

Закрытое наследование целесообразно в том случае, когда меняется сущность нового объекта. Пример:

Базовый класс описывает фигуры на плоскости и имеет методы вычисления площади фигур, а класс-наследник описывает объемные тела, например, призмы с основанием – плоской фигурой, описываемой базовым классом. В этом случае объем тела, описываемого классом-наследником, вычисляется умножением площади основания на высоту. При этом не имеет значения, каким образом получена площадь основания. Кроме того, методы работы с объемными объектами отличны от методов работы с плоскими объектами. Поэтому в данном случае не имеет смысла наследование методов базового класса для работы с объектами, описываемыми классом-наследником:

```
#include <iostream>
using namespace std;
class twom {
        double x,y;
public:
        twom (double x1=1, double y1=1): x(x1), y(y1) {}
        double sq(){  return x*y; }
};
class thm: private twom {
        double z;
public:
        thm(double x1 = 1, double y1 = 1, double z1 = 1):twom(x1,y1),
z(z1){}       double vol(){return sq()*z;}
};

int main(){
        thm t1(1,2,3);
        double d1;
        d1 = t1.vol();
        cout << "vol=  " << d1 << '\n';
return 0;
}
```

Таким образом, закрытое наследование несколько напоминает композицию объектов, когда подобъект находится в закрытой области. Все же необходимо помнить, что наследование – это совсем другая концепция ассоциирования классов, по многим своим свойствам отличная от агрегации, даже в ее строгом варианте (композиции).

5.12 Последовательность создания и уничтожения подобъектов

При наследовании в родительский класс не вносится никаких изменений, вследствие чего при создании объекта на основе класса наследника создается и объект на основе класса родителя.

При создании объекта сначала создается объект на основе родительскго класса (отрабатывает конструктор родительского класса), а затем создается дочерний. При удалении все происходит наоборот, сначала удаляется дочерний, а только потом родительский. Пример.

```
#include <iostream>
using namespace std;
class parent {
public:
   parent() {
      cout<<"Create parent"<<endl;
   }
   ~parent() {
      cout<<"Destroy parent"<<endl;
   }
};
class child:parent {
public:
    child() : parent()     {
       cout<<"Create child"<<endl;
    }
    ~child()     {
       cout<<"Destroy child"<<endl;
    }
};
int main() {
   cout << "Create object" << endl;
   child *ptrChild = new child();
   cout << "Destroy object" << endl;
   delete ptrChild;
   return 0;
}
```

В данном примере описано два класса: parent и child. Класс child является наследником класса parent (квалификатор видимости в данном примере не имеет значения). В конструктор дочернего класса жестко связан с конструктором родительского child() : parent().

В данном примере это не критично, так как в обоих кассах реализовано по одному конструктору. Однако, если конструкторов более одного, бывает необходимо производить их жесткое связывание чтобы дочерний класс отрабатывал как положено.

Результат вывода на экран при выполнении программы:

Create object
Create parent
Create child
Destroy object
Destroy child
Destroy parent

5.13 Виртуальные функции и методы

Полиморфизм времени исполнения обеспечивается за счет использования производных классов и виртуальных функций. Виртуальная функция – это функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или в нескольких производных классах. Виртуальные функции являются особыми функциями, потому что при вызове объекта производного класса с помощью указателя или ссылки на него C++ определяет во время исполнения программы, какую функцию вызвать, основываясь на типе объекта. Для разных объектов вызываются разные версии одной и той же виртуальной функции. Класс, содержащий одну или более виртуальных функций, называется полиморфным классом (polymorphic class).

Виртуальная функция объявляется в базовом классе с использованием ключевого слова virtual. Когда же она переопределяется в производном классе, повторять ключевое слово virtual нет необходимости, хотя и в случае его повторного использования ошибки не возникнет. Пример.

```
#include <iostream>
using namespace std;
class Base {
public:
   virtual void who()    {
      // определение виртуальной функции
      cout << "Class Base" <<endl; }
};
class first_d: public Base {
```

```
public:
   void who() {
      // определение who() применительно к first_d
      cout << "Class first_d"<<endl; }
};
class second_d: public Base {
public:
   void who() {
      // определение who() применительно к second_d
      cout << "Class second_d"<<endl; }
};
int main() {
        Base *p;
        Base base_obj;
        first_d first_obj;
        second_d second_obj;
        p = &base_obj;
        p->who(); // доступ к who класса Base
        p = &first_obj;
        p->who(); // доступ к who класса first_d
        p = &second_obj;
        p->who(); // доступ к who класса second_d
return 0;
}
```

В объекте Base функция who() объявлена как виртуальная. Это означает, что эта функция может быть переопределена в производных классах. В каждом из классов first_d и second_d функция who() переопределена. В функции main() определены три переменные. Первой является объект base_obj, имеющий тип Base. После этого объявлен указатель p на класс Base, затем объекты first_obj и second_obj, относящиеся к двум производным классам. Далее указателю p присвоен адрес объекта base_obji вызвана функция who(). Поскольку эта функция объявлена как виртуальная, то C++ определяет на этапе исполнения, какую из версий функции who() употребить, в зависимости от того, на какой объект указывает указатель p. В данном случае им является объект типа Base, поэтому исполняется версия функции who(), объявленная в классе Base. Затем указателю p присвоен адрес объекта first_obj. (Как известно, указатель на базовый класс может быть использован для любого производного класса.) После того, как

функция who() была вызвана, C++ снова анализирует тип объекта, на который указывает p, для того, чтобы определить версию функции who(), которую необходимо вызвать. Поскольку p указывает на объект типа first_d, то используется соответствующая версия функции who(). Аналогично, когда указателю p присвоен адрес объекта second_obj, то используется версия функции who(), объявленная в классе second_d.

Рассмотрим еще один более наглядный пример:

```
#include <iostream>
using namespace std;
class classVirtual {
public:
   classVirtual() { }
   void f1(void) {
     cout<<"parent f1"<<endl; }

   virtual void f2(void) {
      cout<<"parent f2"<<endl; }
};
class classVirtualChild : public classVirtual {
public:
   classVirtualChild() {  }
   void f1(void) {
      cout<<"child f1"<<endl; }

   void f2(void) {
      cout<<"child f2"<<endl; }
};

int main(int argc, char *argv[])
{
   cout << "Start programm" << endl;

   classVirtual *virt = new classVirtual();
   virt->f1();
   virt->f2();

   cout<<"---------------------------------------"<<endl;

   classVirtualChild *virt2 = new classVirtualChild();
```

```
    virt2->f1();
    virt2->f2();

    cout<<"---------------------------------------"<<endl;

    delete virt;
    virt = new classVirtualChild();
    virt->f1();
    virt->f2();
}
```

В данном примере реалиовано два класса: родиткльский класс classVirtual и дочерний класс classVirtualChild. В обоих классах есть два метода f1() и f2(). В данных методах реализован вывод на экран соответствующей надписи, которая указывает какой метод был вызван и из какого класса. В родительском классе метод f1(void) является обычным, а метод f2(void) является виртуальным.

Результат выполнения программы:

```
parent f1
parent f2
---------------------------------------
child f1
child f2
---------------------------------------
parent f1
child f2
```

Наиболее распространенным способом вызова виртуальной функции служит использование параметра функции. Пример.

```
#include <iostream>
using namespace std;

class Base {
public:
   virtual void who() {
      // определение виртуальной функции
      cout << "Base" << endl;
      }
};
```

```
class first_d: public Base {
public:
   void who () {
      // определение who() применительно к first_d
      cout << "First derivation" << endl;
      }
};

class second_d: public Base {
public:
   void who() {
      // определение who() применительно к second_d
      cout << "Second derivation\n*";
      }
};

// использование в качестве параметра ссылки на базовый класс
void show_who (Base &r) {
   r.who();
}

int main()
{
   Base base_obj;
   first_d first_obj;
   second_d second_obj;
   show_who (base_obj) ; // доступ к who класса Base
   show_who(first_obj); // доступ к who класса first_d
   show_who(second_obj); // доступ к who класса second_d
return 0;
}
```

В данном примере функция show_who() имеет параметр типа ссылки на класс Base. В функции main() вызов виртуальной функции осуществляется с использованием объектов типа Base, first_d и second_d. Вызываемая версия функции who() в функции show_who() определяется типом объекта, на который ссылается параметр при вызове функции.

Ключевым моментом в использовании виртуальной функции для обеспечения полиморфизма времени исполнения служит то, что ис-

пользуется указатель именно на базовый класс. Полиморфизм времени исполнения достигается только при вызове виртуальной функции с использованием указателя или ссылки на базовый класс. Однако ничто не мешает вызывать виртуальные функции, как и любые другие «нормальные» функции, однако достичь полиморфизма времени исполнения на этом пути не удается.

На первый взгляд переопределение виртуальной функции в производном классе выглядит как специальная форма перегрузки функции. Но это не так, и термин перегрузка функции не применим к переопределению виртуальной функции, поскольку между ними имеются существенные различия. Во-первых, функция должна соответствовать прототипу. Как известно, при перегрузке обычной функции число и тип параметров должны быть различными. Однако при переопределении виртуальной функции интерфейс функции должен в точности соответствовать прототипу. Если же такого соответствия нет, то такая функция просто рассматривается как перегруженная и она утрачивает свои виртуальные свойства. Кроме того, если отличается только тип возвращаемого значения, то выдается сообщение об ошибке. (Функции, отличающиеся только типом возвращаемого значения, порождают неопределенность.) Другим ограничением является то, что виртуальная функция должна быть членом, а не дружественной функцией класса, для которого она определена. Тем не менее виртуальная функция может быть другом другого класса.

Если функция была объявлена как виртуальная, то она и остается таковой вне зависимости от количества уровней в иерархии классов, через которые она прошла. Например, если класс second_d получен из класса first_d, а не из класса Base, то функция who() останется виртуальной и будет вызываться корректная ее версия, как показано в следующем примере:

```
// порождение от first_d, а не от Base
class second_d: public first_d {
public:
        void who() {
                // определение who() применительно к second_d
                cout << "Second derivation\n*";
                }
};
```

Если в производном классе виртуальная функция не переопределяется, то тогда используется ее версия из базового класса. Например, запустим следующую версию предыдущей программы:

```
#include <iostream>
using namespace std;

class Base {
public:
   virtual void who() {
   cout << "Base" << endl;
   }
};

class first_d: public Base {
public:
   void who() {
   cout << "First derivation"  << endl;
   }
};

class second_d: public Base {
// who() не определяется
};

int main() {
   Base base_obj;
   Base *p;
   first_d first_obj;
   second_d second_obj;
   p = &base_obj;
   p->who(); // доступ к who класса Base
   p = &first_obj;
   p->who(); // доступ к who класса first_d
   p = &second_obj;
   p->who(); /* доступ к who() класса Base, поскольку second_d не
переопределяет */
return 0;
}
```

Эта программа выдаст следующий результат:

Base
First derivation
Base

Надо иметь в виду, что характеристики наследования носят иерархический характер. Чтобы проиллюстрировать это, предположим, что в предыдущем примере класс second_d порожден от класса first_d вместо класса Base. Когда функцию who() вызывают, используя указатель на объект типа second_d (в котором функция who() не определялась), то будет вызвана версия функции who(), объявленная в классе first_d, поскольку этот класс – ближайший к классу second_d. В общем случае, когда класс не переопределяет виртуальную функцию, C++ использует первое из определений, которое он находит, идя от потомков к предкам.

5.14 Виртуальные деструкторы

В C++ деструктор полиморфного базового класса должен объявляться виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс. Пример.

```
#include <iostream>
using namespace std;

// Вспомогательный класс
class Object {
  public:
    Object() { cout << "Object::ctor()" << endl; }
   ~Object() { cout << "Object::dtor()" << endl; }
};
// Базовый класс
class Base {
  public:
    Base() { cout << "Base::ctor()" << endl; }
    virtual ~Base() { cout << "Base::dtor()" << endl; }
    virtual void print() = 0;
};
// Производный класс
class Derived: public Base {
  public:
```

```
   Derived() { cout << "Derived::ctor()" << endl; }
  ~Derived() { cout << "Derived::dtor()" << endl; }
   void print() {}
   Object  obj;
};
int main () {
   Base * p = new Derived;
   delete p;
   return 0;
}
```

В функции main указателю на базовый класс присваивается адрес динамически создаваемого объекта производного класса Derived. Затем через этот указатель объект разрушается. При этом наличие виртуального деструктора базового класса обеспечивает вызовы деструкторов всех классов в ожидаемом порядке, а именно, в порядке, обратном вызовам конструкторов соответствующих классов.

Вывод данной программы:

```
Base::ctor()
Object::ctor()
Derived::ctor()
Derived::dtor()
Object::dtor()
Base::dtor()
```

Уничтожение объекта производного класса через указатель на базовый класс с невиртуальным деструктором дает неопределенный результат. На практике это выражается в том, что будет разрушена только часть объекта, соответствующая базовому классу. Если в коде выше убрать ключевое слово virtual перед деструктором базового класса, то вывод программы будет уже иным.

В данном случае член данных obj класса Derived также не разрушается:

```
Base::ctor()
Object::ctor()
Derived::ctor()
Base::dtor()
```

Если базовый класс предназначен для полиморфного использования, то его деструктор должен объявляться виртуальным. Для реализации механизма виртуальных функций каждый объект класса хранит указатель на таблицу виртуальных функций vptr, что увеличивает

его общий размер. Обычно, при объявлении виртуального деструктора такой класс уже имеет виртуальные функции, и увеличения размера соответствующего объекта не происходит.

Если же базовый класс не предназначен для полиморфного использования (не содержит виртуальных функций), то его деструктор не должен объявляться виртуальным.

6. Перегрузка операций

Примером полиморфизма в языке C++ является перегрузка (overload) операций. Она позволяет манипулировать объектами классов используя обычный синтаксис языка C. Для обеспечения такой возможности в перегружаемых операциях должно сохраняться количество аргументов соответствующей операции. Например, операция деления, обозначаемая "/", в C имеет два аргумента, поэтому ерегружаемая одноименная операция "/" также должна принимать два аргумента. Как правило, в качестве аргументов перегружаемых операций и возвращаемых ими значений выступают собственные классы. Так, если перегрузить операцию сложения для класса complex, то можно использовать естественный синтаксис языка C для их сложения:

```
class complex     {
double re, im;
public:
. . . . . .
}
. . . . . .
complex a, b, c;
. . . . . .
c = a + b;
```

Все перегружаемые операции имеют тот же приоритет и правила ассоциативности, что и предопределенные операции языка. По этой причине их можно использовать для построения цепочек операций, например:

```
x = a + b + c + d;
```

Отметим, что в C++ нельзя перегрузить операции . :: .* и ?. Язык C++ не позволяет отличать постфиксную и префиксную формы перегруженных операций, поэтому, например, для перегруженной операции ++ выражения

```
++x и x++
```

имеют одно и то же значение. В общем случае предполагается, что перегруженные унарные операции используются в префиксной форме, поэтому для выражения "x++" компилятор выдаст предупреждение. Нельзя также определять новые лексические символы операций.

Язык C++ обеспечивает достаточную гибкость в перегрузке операций и их наследовании. Так, все перегруженные операции базовых классов, за исключением операции присваивания, наследуются производным классом. Перегруженная операция базового класса может затем вновь перегружаться в производном классе и т.д. Операции перегружаются с помощью функций-операторов, имеющих следующий формат:

<тип> operator <символ> (<список_аргументов>) {...}

где тип - это тип возвращаемого операцией значения, а символ - символьное обозначение перегруженной операции.

Функции-операторы могут вызываться таким же образом, как и любая другая функция. Использование операции - это лишь сокращенная запись явного вызова функцииоператора, например для комплексных чисел сокращенная запись

c = a + b;

эквивалентна вызову функции

c = operator+(a,b);

В отношении действий, выполняемых перегружаемыми операциями, C++ не накладывает никаких ограничений. Можно, например, перегрузить операцию "+" для выполнения вычитания комплексных чисел, а "-" - для их сложения, но подобное использование механизма перегрузки операций не рекомендуется с точки зрения здравого смысла.

Сложные операции C++, равносильные комбинации нескольких операций, автоматически не раскрываются и, если пользователь их не определил, остаются компилятору неизвестными. Например, если для комплексных чисел перегружены операции сложения "+" и присваивания "=", то запись вида

a += b;

совсем не означает, что будет выполняться

a = a + b;

Чтобы можно было применять операцию "+=" с комплесными числами, она должна быть соответствующим образом перегружена в классе complex.

Функции-операторы, являющиеся "друзьями" класса, всегда имеют число аргументов, равное числу аргументов перегружаемой

операции. Для унарных операций - это один аргумент, для бинарных - два и т.д.

```
class complex
{
        double re, im;
public:
        complex (double r=0, double i=0)
                { re = r; im = i; }
        void print(char *s)
                {...);
}

friend complex operator + (complex, complex);
};

complex operator + (complex a, complex b)
{ return complex(a.re + b.re, a.im + b.im); }
```

или :

```
class complex{...}
complex operator + (complex a, complex b)
{
        complex c;
        c.re = a.re + b.re;
        c.im = a.im + b.im;
return c;
}
```

Теперь для сложения комплексных чисел можно использовать естественный синтаксис языка C++:

```
        complex x(2.0,1.0), y(3.14,1.0), z;
        z = x + y;
```

Если попытаться для класса complex аналогичным образом перегрузить унарную операцию инкремента следующим образом :

```
class complex
{
        . . . . . .
public:
```

```
        . . . . . .
friend complex operator ++(complex);
};
complex operator ++(complex a)
        { return complex(a.re++, a.im++); }
```

или :

```
class complex{...};
complex operator ++(complex a)
{
        a.re++,
        a.im++;
return a;
}
```

то можно убедиться, что перегруженная операция работать не будет. Дело в том, что в С++ аргументы передаются по значению и, следовательно, возвращаемая величина не изменяетсвоего значения. Можно использовать в качестве аргумента указатель :

```
class complex {
        . . . . . .
public:
        friend complex operator ++ (complex *);
};
complex operator + (complex *p) {
        p->re++,
        p->im++;
return *p;
}
```

Однако в этом случае компилятор выдаст ошибку, поскольку язык С++ требует, чтобы операнд операции "++" имел тип класса объекта. Выходом из этой ситуации является использование ссылок в качестве аргумента функции-оператора:

```
class complex {
        . . . . . .
public:
friend complex operator ++(complex &);
};
complex operator + (complex &r) {
        r.re++,
```

```
        r.im++;
return r;
}
```

Невозможность перегрузки унарных операций функциями "друзьями" послужило одной из причин введения ссылок в язык C++. Ссылки в качестве аргументов перегружаемых операций позволяют передавать реализующей ее функции не копии объектов, а их адреса, что позволяет изменять значения этих аргументов. Кроме того, использование ссылок в качестве аргументов для больших объектов позволяет значительно сократить объем копируемой информации, что приводит к повышению эффективности программы.

Рассмотрим теперь вопрос каким должно быть возвращаемое значение функцииоператора. Оно не может быть ссылкой на автоматическую переменную, так как после выхода из функции возвращаемое значение указывает на несуществующую переменную. Оно также не может быть статической переменной. Допускается ссылка на переменную в "куче", однако это усложняет программирование. Лучшим решением с точки зрения простоты реализации и эффективности считается копирование возвращаемого значения . Можно также возвращать ссылку, но только на существующий объект, например, когда эта ссылка передается функции-оператору в качестве параметра:

```
class complex {
        . . . . . .
public:
        . . . . . .
friend complex& operator ++(complex& r);
};
complex& operator ++(complex& r)
{
        r.re++;
        r.im++;
return r;
}
```

Для функций-операторов, являющихся компонентами класса, первым аргументом по умолчанию является указатель this на тот объект, к которому она относится. Поэтому число аргументов для таких операторов на единицу меньше, по сравнению с числом аргументов перегружаемой операции. Например, для бинарных операций - один аргумент, для унарных - аргументы отсутствуют и т.д. Перегрузка

операций + и ++ для класса complex, реализованная с помощью операторов компонент имеет вид:

```
class complex {
        . . . . . .
public:
        complex operator + (complex b);
        complex operator ++ (void);
};
        complex complex:: operator + (complex b)
                { return complex(re + b.re, im + b.im); };
        complex complex:: operator ++()
                { return complex(re++, im++); };
```

Отметим, что при перегрузке операций с помощью компонент функций широко используется указатель this. Однако при этом следует соблюдать особую осторожность. Соблазнительной может показастья идея использовать для получения результата вместо промежуточной переменной первый аргумент, например:

```
class complex {
. . . . . .
public:
        complex operator +(complex b);
        complex operator ++(void);
};
complex complex:: operator + (complex b)
{
        re += b.re;
        im += b.im;
return *this;
};
complex complex:: operator ++ ()
{
        re++;
        im++;
return *this;
};
```

В последнем примере операция инкремента работает правильно, а операция сложения имеет побочный эффект, увеличивая значение первого операнда.

Напомним, что с помощью функций-операторов, являющихся компонентами класса, можно перегружать операции =, (), [] и ->, которые запрещены для перегрузки функциями "друзьями".

Может возникнуть вопрос в каких случаях при перегрузке операций следует использовать функции-"друзья", а в каких - функции-компоненты? Рассмотрим пример:

```
main(void)
{
        complex a(2.0,1.0), b(3.14,1.0), c;
        double d = 1.5;
        c = a + b;
        c = a + d;
        c = d + b;
        . . . . .
}
```

Выражение c = a + b будет верным, поскольку a и b оба типа complex, для которого переопределена операция "+". Выражение c = a + d также будет верным, поскольку оно интерпретируется как a.operator+(complex(d)) Однако выражение c = d + b вызовет ошибку, если операция "+" перегружена с помощью функции-компоненты. Дело в том, что d + b по определению эквивалентно d.operator+(b), но d не является объектом класса и не имеет компонент! В случае же перегрузки "+" как функции-друга выражение верно, поскольку здесь будет вызван конструктор инициализации и d + b будет эквиваленто complex(d) + b Отсюда можно сделать следующий вывод: в том случае, если предполагается неявное преобразование операндов перегружаемой операции, то реализующую ее функцию лучше сделать "другом". К таким операциям можно отнести арифметические и логические операции (+, -, || и т. д.).

Если же операция изменяет состояние объекта, а это можно сделать только для ранее созданного объекта, то реализующая операцию функция должна быть компонентой класса. К подобным операциям относятся =, *=, ++ и др.

При прочих равных условиях для перегрузки операций чаще используются функции-компоненты. Это связано с тем, что они имеют на один аргумент меньше, допускают использование неявного параметра this, не чувствительны к модификациям функций преобразования типа и, наконец, описание функции-компоненты, как правило, короче описания функции-друга.

6.1 Многократная перегрузка операций

C++ позволяет несколько раз перегружать одну и ту же операцию в пределах одного класса. Важно только, чтобы однозначно определялся вызов необходимой операции, т.е. функции-операторы должны различаться количеством или типом своих аргументов, Например:

```
class String {
        char *str;
public:
        String (char * s = "\0")
        // конструктор
        { str = new char [strlen (s) +1]; strcpy (str,s); }
        String operator + (String t)
        // первая перзагрузка +
        { return String(strcat(str,t.str)); }
        String operator + (char *s)
        // вторая перзагрузка +
        { return String(strcat(str,s)); }
        void print(char* s)
        { cout << s << ": " << str << "\n"; }
};
main(void) {
String a("Минск"), b(" - город"), c;
c = a + b + " -герой!";
c.print("c");
}
```

В данном примере дважды перегружена операция + для класса String. В первом случае она позволяет нам связывать объекты типа String между собой, а во втором - тип String со строкой символов. Это дает возможность в одном выражении записывать как объекты типа String, так и обычные строки типа (char *).

6.2 Перегрузка операции присваивания

Операция присваивания перегружается обычным образом. Особенностью этой операции является то, что она не может быть унаследована производным классом. Кроме того, в C++ операцию "=" можно перегружать только с помощью функции-компоненты.

Например:

```
class complex {
        . . . . . .
```

```
public:
        complex& operator =(complex &);
};
complex& complex:: operator =(complex& b)
{
        re = b.re;
        im = b.im;
return *this;
}
```

Здесь первый операнд передается через указатель this, принимает значение второго операнда, полученного в качестве аргумента, и возвращается с помощью выражения *this.

Не смотря на то, что в этом примере изменяется значение первого аргумента, это не является ошибкой, поскольку его значение как раз и нужно изменить. Данный пример также демонстрирует те немногие случаи, когда в программе необходимо явно использовать указатель this.

Если для некоторого класса X операция присваивания не определена, то при необходимости она формируется компилятором C++ автоматически в виде:

```
X & X:: operator = (const X & <источник>) {
        // покомпонентное присваивание
return *this;
}
```

Рассмотрим на примере класса String более сложный случай перегрузки операции "=", чем простое покомпонентное присваивание. Первый вариант перегрузки операции "=" для класса String на первый взгляд вообще не возвращает никакого значения

```
class String {
        . . . . . .
public:
void operator = (Srting &t){
        if (this == &t)
                return;
        // копирование в ту же строку
        delete str;
        // удаляется старая строка
        str = new char[strlen(t.str)+1]; // образуется новая
        strcpy(str,t.str);
```

```
        // в нее копируется значение
        }
}
```

Здесь вначале проверяется, чтобы строка не дублировала саму себя. Затем уничтожается старое значение строки, отводится память необходимого объема и туда копируется принимаемый аргумент.

Более корректной реализацией перегрузки операции "=" будет явный возврат сформированного объекта с помощью указателя this:

```
class String {
        . . . . . .
public:
        String& operator = (Srting &t)
        {
                if (this == &t)
                        return *this;
        delete str;
        // удаляется старая строка
        str = new char[strlen(t.str)+1]; // образуется новая
        strcpy(str,t.str);
        // в нее копируется значение
        return *this;
        }
}
```

Если сравнить последнюю реализацию с конструктором класса String, то можно увидеть их значительное сходство. Поэтому можно воспользоваться уже готовым конструктором для формирования возвращаемого значения:

```
class String {
        . . . . . .
public:
String operator = (Srting &t)
{
        if (this == &t) return *this;
        delete str;
        // удаляется старая строка
return String(t.str);
}
}
```

Заметим, что здесь изменился тип возвращаемого значения, поскольку конструктор String::String() возвращает объект, а не ссылку на него. Таким образом, для простых типов (например, complex) операцию присваивания можно не перегружать, в этом случае она формируется компилятором автоматически, осуществляя покомпонентное присваивание. Если объекты сложные и покомпонентное присваивание для них не работает, следует операцию присваивания перегружать явно. В качестве возвращаемого значения перегруженной операции присваивания чаще используется ссылка на объект . Возможно в возвращаемом выражении обращаться к конструктору класса. В последнем случае возвращаемым значением будет сам объект.

6.3 Перегрузка операций [], () и ->

Прежде всего отметим, что эти операции перегружаются только с помощью компонент-функций и их нельзя перегрузить функциями-"друзьями".

Начнем с рассмотрения перегрузки операции []. Предположим, что мы хотим обращаться к действительной и мнимой частям некоторого комплексного числа h как к элементам массива, т. е. h.re соответствует h[0] и h.im соответствует h[1]. Для этого можно перегрузить операцию [] выделения элемента массива для класса complex следующим образом :

```
class complex {
        double re, im;
public:
        . . . . . .
        double & operator [] (int i)
                { return *(&re + i);}
};

main(void) {
        complex one(1.0, 0.0);
        cout << "one.re=" << one[0] << "one.im=" << one[1] << "\n";
}
```

Перегрузка операции [] для класса X позволяет всякое выражение вида ob[i], где ob – объект класса X, интерпретировать как обращение к функции ob.operator[](i). Операция вызова функции вида:

<имя>(<список_аргументов>);

в языке C++ рассматривается как бинарная операция, где первым операндом является <имя>, а вторым – <список_аргументов>. При перегрузке операции () список аргументов вычисляется и проверяется в соответствии с обычными правилами передачи аргументов. В общем случае список аргументов может быть пуст.

Перегрузка операции () позволяет рассматривать выражение вида
ob(<список_аргументов>)
как обращение к функции
ob.operator()(<список_аргументов>).

Рассмотрим перегрузку операции () для класса Array. Иногда целесообразно считать, что индекс массива начинается не с нуля, а с единицы (подобно тому, как это реализовано в языке PL/1). Для этого перегрузим операцию () :

```
class Array {
        . . . . . .
public:
        int operator()(int i) { return m[i-1]; }
};
void main() {
        Array x(5);
        cout << "x= ";
        for (int i=1; i<=5; i++)
        cout << x(i) << " ";
        cout << endl;
}
```

Операцию () обычно перегружают для операций, требующих большого числа операндов, для классов с единственно возможной операцией, а также когда некоторая операция используется особенно часто. Перегружаемая операция () не может быть статической компонентой-функцией класса.

C++ предоставляет пользователю возможность выполнять некоторую предварительную обработку до обращения к компонентам классов. С этой целью используется перегузка операций ->, * и &. Операция → рассматривается как унарная операция и ее перегрузка позволяет выражение вида
ob->m
трактовать как
(ob.operator->())->m.

Причем функция-оператор, реализующая операцию -> должна либо возвращать указатель на объект данного класса, либо возвращать объект этого класса, т. е. ее описание для класса X должно иметь вид:

```
X *operator -> () {...}
```

или

```
X operator -> () {...}.
```

Рассмотрим случай, когда возвращаемым значением перегуженной операции -> является тип этого же класса:

```
class X {
public:
        int a;
        X(int i) { a = i; }
        X operator ->()
        {
        cout << "Доступ к компонентам класса X\n";
return *this;
}
};
main(void) {
        X x(5);
        X *px = new X(10);
        cout << "a= " << x->a << "\n"; // перегруженная операция
        cout << "a= " << px->a << "\n"; // предопределенная операция
}
```

В данном примере перед обращением к компоненте в стандартный поток выводится сообщение.Наибольший интерес представляет случай, когда возвращаемым значением перегруженной операции -> класса X является указатель на некоторый другой класс Y. В этом случае операция -> вначале применяется к своему левому операнду для получения указателя p на класс Y, а затем указатель p используется как левый операнд бинарной операции -> для доступа к компоненте класса Y, например:

```
class Y {
public:
        int b;
        Y(int j) { b = j; }
};
class X {
```

```
        int a;
public:
        Y *p;
        X(int i, int j)
{
        a = i;
        p = new Y(j);
}
Y* operator ->()
{
        cout << "Доступ к компонентам класса Y\n";
return p;
}
};
main(void) {
        X x(3,5);
        cout << "b= " << x->b << "\n"; // перегруженная операция
        cout << "b= " << x.p->b << "\n"; // базовая операция
}
```

Если класс Y в свою очередь имеет перегруженную операцию ->, то p будет использован в качестве левого операнда унарной опеации -> и вся процедура повторится для класса Y.

Унарные операции * и & перегружаются аналогично, причем сохраняется соответствующая семантика между операцией -> и операциями * и &.

7. Обработка исключительных ситуаций

Обработка исключений – это механизм, позволяющий двум независимо разработанным программным компонентам взаимодействовать в аномальной ситуации, называемой исключением.

7.1 Возбуждение исключения

Исключение – это аномальное поведение во время выполнения, которое программа может обнаружить, например: деление на 0, выход за границы массива или истощение свободной памяти. Такие исключения нарушают нормальный ход работы программы, и на них нужно немедленно отреагировать. В C++ имеются встроенные средства для

их возбуждения и обработки. С помощью этих средств активизируется механизм, позволяющий двум несвязанным (или независимо разработанным) фрагментам программы обмениваться информацией об исключении.

Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или возбудить, исключение. Чтобы понять, как это происходит, реализуем класс iStack, используя исключения для извещения об ошибках при работе со стеком. Определение класса

```
#include
class iStack {
public:
  iStack( int capacity )
      : _stack( capacity ), _top( 0 ) { }

  bool pop( int &top_value );
  bool push( int value );

  bool full();
  bool empty();
  void display();

  int size();
private:
  int _top;
  vector< int > _stack;
};
```

Стек реализован на основе вектора из элементов типа int. При создании объекта класса iStack его конструктор создает вектор из int, размер которого (максимальное число элементов, хранящихся в стеке) задается с помощью начального значения. Например, следующая инструкция создает объект myStack, который способен содержать не более 20 элементов типа int:

```
iStack myStack(20);
```

При манипуляциях с объектом myStack могут возникнуть две ошибки:

- запрашивается операция pop(), но стек пуст;
- запрашивается операция push(), но стек полон.

Вызвавшую функцию нужно уведомить об этих ошибках посредством исключений. Во-первых, должны быть определены, какие именно исключения могут быть возбуждены. В C++ они чаще всего реализуются с помощью классов. Эти определения мы поместим в заголовочный файл stackExcp.h:

```
// stackExcp.h
class popOnEmpty { /* ... */ };
class pushOnFull { /* ... */ };
```

Затем надо изменить определения функций-членов pop() и push() так, чтобы они возбуждали эти исключения. Для этого предназначена инструкция throw, которая во многих отношениях напоминает return. Она состоит из ключевого слова throw, за которым следует выражение того же типа, что и тип возбуждаемого исключения.

Попробуем такой вариант:

```
throw popOnEmpty;
```

Исключение – это объект, и функция pop() должна генерировать объект класса соответствующего типа. Выражение в инструкции throw не может быть просто типом. Для создания нужного объекта необходимо вызвать конструктор класса. Инструкция throw для функции pop() будет выглядеть так:

```
// инструкция является вызовом конструктора
throw popOnEmpty();
```

Эта инструкция создает объект исключения типа popOnEmpty.

Методы pop() и push() были определены как возвращающие значение типа bool: true означало, что операция завершилась успешно, а false – что произошла ошибка. Поскольку теперь для извещения о неудаче pop() и push() используют исключения, возвращать значение необязательно. Поэтому мы будем считать, что эти методы имеют тип void:

```
class iStack {
public:
  // ...
  // больше не возвращают значения
  void pop( int &value );
  void push( int value );
private:
  // ...
};
```

Теперь функции, пользующиеся нашим классом iStack, будут предполагать, что все хорошо, если только не возбуждено исключение; им больше не надо проверять возвращенное значение, чтобы узнать, как завершилась операция.

```
#include "stackExcp.h"
void iStack::pop( int &top_value )
{
  if ( empty() )
                throw popOnEmpty();
                top_value = _stack[ --_top ];
                cout << "iStack::pop(): "<<top_value " endl;
}
void iStack::push( int value )
{
        cout << "iStack::push( "<< value << " )\n";
  if ( full() )
        throw pushOnFull( value );
        stack[ _top++ ] = value;
}
```

Хотя исключения чаще всего представляют собой объекты типа класса, инструкция throw может генерировать объекты любого типа. Например, функция mathFunc() в следующем примере возбуждает исключение в виде объекта-перечисления:

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };
int mathFunc( int i ) {
  if ( i == 0 )
     throw zeroOp;  // исключение в виде объекта-перечисления

  // в противном случае продолжается нормальная обработка
}
```

Try-блок

```
#include
#include "iStack.h"

int main() {
  iStack stack( 32 );

 stack.display();
```

```
 for ( int ix = 1; ix < 51; ++ix )
 {
        if ( ix % 3 == 0 )
              stack.push( ix );

        if ( ix % 4 == 0 )
          stack.display();

        if ( ix % 10  == 0 ) {
        int dummy;
        stack.pop( dummy );
        stack.display();
        }
 }
return 0;
}
```

В данной программе тестируется определенный в предыдущем разделе класс iStack и его методы pop() и push(). Выполняется 50 итераций цикла for. На каждой итерации в стек помещается значение, кратное 3: 3, 6, 9 и т.д. Если значение кратно 4 (4, 8, 12...), то выводится текущее содержимое стека, а если кратно 10 (10, 20, 30...), то с вершины снимается один элемент, после чего содержимое стека выводится снова.

Инструкции, которые могут возбуждать исключения, должны быть заключены в try-блок. Такой блок начинается с ключевого слова try, за которым идет последовательность инструкций, заключенная в фигурные скобки, а после этого – список обработчиков, называемых catch-предложениями. Try-блок группирует инструкции программы и ассоциирует с ними обработчики исключений.

```
for ( int ix = 1; ix < 51; ++ix ) {
try {   // try-блок для исключений pushOnFull
        if ( ix % 3 == 0 )
              stack.push( ix );
 }
catch ( pushOnFull ) { ... }
if ( ix % 4 == 0 )
       stack.display();

try {   // try-блок для исключений popOnEmpty
```

```
        if ( ix % 10  == 0 ) {
         int dummy;
        stack.pop( dummy );
        stack.display();
        }
}
catch ( popOnEmpty ) { ... }
}
```

В таком виде программа выполняется корректно. Однако обработка исключений в ней перемежается с кодом, использующимся при нормальных обстоятельствах, а такая организация несовершенна. В конце концов, исключения – это аномальные ситуации, возникающие только в особых случаях. Желательно отделить код для обработки аномалий от кода, реализующего операции со стеком. Следующая схема облегчает чтение и сопровождение программы:

```
try {
for ( int ix = 1; ix < 51; ++ix )
{
        if ( ix % 3 == 0 )
                stack.push( ix );

        if ( ix % 4 == 0 )
                stack.display();

        if ( ix % 10  == 0 ) {
                int dummy;
                stack.pop( dummy );
                stack.display();
        }
        }
}
catch ( pushOnFull ) { ... }
catch ( popOnEmpty ) { ... }
```

С try-блоком ассоциированы два catch-предложения, которые могут обработать исключения pushOnFull и popOnEmpty, возбуждаемые функциями-членами push() и pop() внутри этого блока. Каждый catch-обработчик определяет тип "своего" исключения. Код для обработки исключения помещается внутрь составной инструкции (между

фигурными скобками), которая является частью catch-обработчика. (Подробнее catch-предложения мы рассмотрим в следующем разделе.)

Исполнение программы может пойти по одному из следующих путей:

– если исключение не возбуждено, то выполняется код внутри try-блока, а ассоциированные с ним обработчики игнорируются. Функция main() возвращает 0;

– если функция-член push(), вызванная из первой инструкции if внутри цикла for, возбуждает исключение, то вторая и третья инструкции if игнорируются, управление покидает цикл for и try-блок, и выполняется обработчик исключений типа pushOnFull;

– если функция-член pop(), вызванная из третьей инструкции if внутри цикла for, возбуждает исключение, то вызов display() игнорируется, управление покидает цикл for и try-блок, и выполняется обработчик исключений типа popOnEmpty.

Когда возбуждается исключение, пропускаются все инструкции, следующие за той, где оно было возбуждено. Исполнение программы возобновляется в catch-обработчике этого исключения. Если такого обработчика не существует, то управление передается в функцию terminate(), определенную в стандартной библиотеке C++. Try-блок может содержать любую инструкцию языка C++: как выражения, так и объявления. Он вводит локальную область видимости, так что объявленные внутри него переменные недоступны вне этого блока, в том числе и в catch-обработчиках. Например, функцию main() можно переписать так, что объявление переменной stack окажется в try-блоке. В таком случае обращаться к этой переменной в catch-обработчиках нельзя:

```
int main() {
try {
        iStack stack( 32 );     // правильно: объявление внутри try-блока

        stack.display();
         for ( int ix = 1; ix < 51; ++ix )
        {
                // то же, что и раньше
        }
}
catch ( pushOnFull ) {
```

```
        // здесь к переменной stack обращаться нельзя
        }
    catch ( popOnEmpty ) {
        // здесь к переменной stack обращаться нельзя
         }

     // и здесь к переменной stack обращаться нельзя
    return 0;
    }
```

Можно объявить функцию так, что все ее тело будет заключено в try-блок. При этом не обязательно помещать try-блок внутрь определения функции, удобнее заключить ее тело в функциональный try-блок. Такая организация поддерживает наиболее чистое разделение кода для нормальной обработки и кода для обработки исключений. Например:

```
    int main()
    try {
        iStack stack( 32 );      // правильно: объявление внутри try-блока

        stack.display();
         for ( int ix = 1; ix < 51; ++ix )
        {
        // то же, что и раньше
        }

     return 0;
    }
    catch ( pushOnFull ) {
        // здесь к переменной stack обращаться нельзя
    }
    catch ( popOnEmpty ) {
        // здесь к переменной stack обращаться нельзя
    }
```

Ключевое слово try находится перед фигурной скобкой, открывающей тело функции, а catch-обработчики перечислены после закрывающей его скобки. Как видим, код, осуществляющий нормальную обработку, находится внутри тела функции и четко отделен от

кода для обработки исключений. Однако к переменным, объявленным в main(), нельзя обратиться из обработчиков исключений.

Функциональный try-блок ассоциирует группу catch-обработчиков с телом функции. Если инструкция возбуждает исключение, то поиск обработчика, способного перехватить это исключение, ведется среди тех, что идут за телом функции. Функциональные try-блоки особенно полезны в сочетании с конструкторами классов.

7.2 Перехват исключений

В языке C++ исключения обрабатываются в предложениях catch. Когда какая-то инструкция внутри try-блока возбуждает исключение, то просматривается список последующих предложений catch в поисках такого, который может его обработать.

Catch-обработчик состоит из трех частей: ключевого слова catch, объявления одного типа или одного объекта, заключенного в круглые скобки (оно называется объявлением исключения), и составной инструкции. Если для обработки исключения выбрано некоторое catch-предложение, то выполняется эта составная инструкция. Рассмотрим catch-обработчики исключений pushOnFull и popOnEmpty в функции main() более подробно:

```
catch ( pushOnFull ) {
        cerr << "trying to push value on a full stack\n";
        return errorCode88;
}
catch ( popOnEmpty ) {
        cerr << "trying to pop a value on an empty stack\n";
        return errorCode89;
}
```

В обоих catch-обработчиках есть объявление типа класса; в первом это pushOnFull, а во втором – popOnEmpty. Для обработки исключения выбирается тот обработчик, для которого типы в объявлении исключения и в возбужденном исключении совпадают. (В главе 19 мы увидим, что типы не обязаны совпадать точно: обработчик для базового класса подходит и для исключений с производными классами.) Например, когда функция-член pop() класса iStack возбуждает исключение popOnEmpty, то управление попадает во второй обработчик. После вывода сообщения об ошибке в cerr, функция main() возвращает код errorCode89.

После завершения обработчика выполнение возобновляется с инструкции, идущей за последним catch-обработчиком в списке. В нашем примере оно продолжается с инструкции return в функции main(). После того как catch-обработчик popOnEmpty выведет сообщение об ошибке, main() вернет 0.

```
int main() {
        iStack stack( 32 );

       try {
       stack.display();
        for ( int x = 1; ix < 51; ++ix )
       {
              // то же, что и раньше
       }
}
catch ( pushOnFull ) {
       cerr << "trying to push value on a full stack\n";
}
 catch ( popOnEmpty ) {
       cerr << "trying to pop a value on an empty stack\n";
}

// исполнение продолжается отсюда
 return 0;
}
```

Говорят, что механизм обработки исключений в C++ невозвратный: после того как исключение обработано, управление не возобновляется с того места, где оно было возбуждено. В нашем примере управление не возвращается в функцию-член pop(), возбудившую исключение.

7.3 Объекты-исключения

Объявлением исключения в catch-обработчике могут быть объявления типа или объекта. В каких случаях это следует делать? Тогда, когда необходимо получить значение или как-то манипулировать объектом, созданным в выражении throw. Если классы исключений спроектированы так, что в объектах-исключениях при возбуждении сохраняется некоторая информация и если в объявлении исключения

фигурирует такой объект, то инструкции внутри catch-обработчика могут обращаться к информации, сохраненной в объекте выражением throw.

Изменим реализацию класса исключения pushOnFull, сохранив в объекте-исключении то значение, которое не удалось поместить в стек. Catch-обработчик, сообщая об ошибке, теперь будет выводить его в cerr. Для этого мы сначала модифицируем определение типа класса pushOnFull следующим образом:

```
// новый класс исключения:
// он сохраняет значение, которое не удалось поместить в стек
class pushOnFull {
public:
        pushOnFull( int i ) : _value( i ) { }
        int value { return _value; }
private:
        int _value;
};
```

Новый закрытый член _value содержит число, которое не удалось поместить в стек. Конструктор принимает значение типа int и сохраняет его в члене _data. Вот как вызывается этот конструктор для сохранения значения из выражения throw:

```
void iStack::push( int value )
{
        if ( full() )
                // значение, сохраняемое в объекте-исключении
        throw pushOnFull( value );

        // ...
}
```

У класса pushOnFull появилась также новая функция-член value(), которую можно использовать в catch-обработчике для вывода хранящегося в объекте-исключении значения:

```
catch ( pushOnFull eObj ) {
  cerr << "trying to push value << "eObj.value()
     << "on a full stack\n";
}
```

Обратите внимание, что в объявлении исключения в catch-обработчике фигурирует объект eObj, с помощью которого вызывается функция-член value() класса pushOnFull.

Объект-исключение всегда создается в точке возбуждения, даже если выражение throw – это не вызов конструктора и, на первый взгляд, не должно создавать объекта.Например:

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };
enum EHstate state = noErr;
int mathFunc( int i ) {
        if ( i == 0 ) {
                state = zeroOp;
                throw state;    // создан объект-исключение
        }
// иначе продолжается обычная обработка
}
```

В этом примере объект state не используется в качестве объекта-исключения. Вместо этого выражением throw создается объект-исключение типа EHstate, который инициализируется значением глобального объекта state. Как программа может различить их? Для ответа на этот вопрос мы должны присмотреться к объявлению исключения в catch-обработчике более внимательно.

Это объявление ведет себя почти так же, как объявление формального параметра. Если при входе в catch-обработчик исключения выясняется, что в нем объявлен объект, то он инициализируется копией объекта-исключения. Например, следующая функция calculate() вызывает определенную выше mathFunc(). При входе в catch-обработчик внутри calculate() объект eObj инициализируется копией объекта-исключения, созданного выражением throw.

```
void calculate( int op ) {
        try {
                mathFunc( op );
        }
catch ( EHstate eObj ) {
        // eObj - копия сгенерированного объекта-исключения
        }
}
```

Объявление исключения в этом примере напоминает передачу параметра по значению. Объект eObj инициализируется значением объекта-исключения точно так же, как переданный по значению фор-

мальный параметр функции – значением соответствующего фактического аргумента.

Как и в случае параметров функции, в объявлении исключения может фигурировать ссылка. Тогда catch-обработчик будет напрямую ссылаться на объект-исключение, сгенерированный выражением throw, а не создавать его локальную копию:

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate &eObj ) {
        // eObj ссылается на сгенерированный объект-исключение
    }
}
```

Для предотвращения ненужного копирования больших объектов применять ссылки следует не только в объявлениях параметров типа класса, но и в объявлениях исключений того же типа.

В последнем случае catch-обработчик сможет модифицировать объект-исключение. Однако переменные, определенные в выражении throw, остаются без изменения. Например, модификация eObj внутри catch-обработчика не затрагивает глобальную переменную state, установленную в выражении throw:

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate &eObj ) {
        // исправить ошибку, вызвавшую исключение
        eObj = noErr;  // глобальная переменная state не изменилась
    }
}
```

Catch-обработчик переустанавливает eObj в noErr после исправления ошибки, вызвавшей исключение. Поскольку eObj – это ссылка, можно ожидать, что присваивание модифицирует глобальную переменную state. Однако изменяется лишь объект-исключение, создан-

ный в выражении throw, поэтому модификация eObj не затрагивает state.

7.4 Раскрутка стека

Поиск catch-обработчикадля возбужденного исключения происходит следующим образом. Когда выражение throw находится в try-блоке, все ассоциированные с ним предложения catch исследуются с точки зрения того, могут ли они обработать исключение. Если подходящее предложение catch найдено, то исключение обрабатывается. В противном случае поиск продолжается в вызывающей функции. Предположим, что вызов функции, выполнение которой прекратилось в результате исключения, погружен в try-блок; в такой ситуации исследуются все предложения catch, ассоциированные с этим блоком. Если один из них может обработать исключение, то процесс заканчивается. В противном случае переходим к следующей по порядку вызывающей функции. Этот поиск последовательно проводится во всей цепочке вложенных вызовов. Как только будет найдено подходящее предложение, управление передается в соответствующий обработчик.

В нашем примере первая функция, для которой нужен catch-обработчик, – это метод pop() класса iStack. Поскольку выражение throw внутри pop() не находится в try-блоке, то программа покидает pop(), не обработав исключение. Следующей рассматривается функция, вызвавшая pop(), то есть main(). Вызов pop() внутри main() находится в try-блоке, и далее исследуется, может ли хотя бы одно ассоциированное с ним предложение catch обработать исключение. Поскольку обработчик исключения popOnEmpty имеется, то управление попадает в него.

Процесс, в результате которого программа последовательно покидает составные инструкции и определения функций в поисках предложения catch, способного обработать возникшее исключение, называется раскруткой стека. По мере раскрутки прекращают существование локальные объекты, объявленные в составных инструкциях и определениях функций, из которых произошел выход. C++ гарантирует, что во время описанного процесса вызываются деструкторы локальных объектов, хотя они исчезают из-за возбужденного исключения.

Если в программе нет предложения catch, способного обработать исключение, оно остается необработанным. Но исключение – это настолько серьезная ошибка, что программа не может продолжать выполнение. Поэтому, если обработчик не найден, вызывается функция

terminate() из стандартной библиотеки C++. По умолчанию terminate() активизирует функцию abort(), которая аномально завершает программу.

Выражение throw ведет себя аналогично вызову, а предложение catch чем-то напоминает определение функции. Основная разница между этими двумя механизмами заключается в том, что информация, необходимая для вызова функции, доступна во время компиляции, а для обработки исключений – нет. Обработка исключений в C++ требует языковой поддержки во время выполнения. Например, для обычного вызова функции компилятору в точке активизации уже известно, какая из перегруженных функций будет вызвана. При обработке же исключения компилятор не знает, в какой функции находится catch-обработчик и откуда возобновится выполнение программы. Функция terminate() предоставляет механизм времени выполнения, который извещает пользователя о том, что подходящего обработчика не нашлось.

7.5 Повторное возбуждение исключения

Может оказаться так, что в одном предложении catch не удалось полностью обработать исключение. Выполнив некоторые корректирующие действия, catch-обработчик может решить, что дальнейшую обработку следует поручить функции, расположенной "выше" в цепочке вызовов. Передать исключение другому catch-обработчику можно с помощью повторного возбуждения исключения. Для этой цели в языке используется конструкция:

```
throw;
```

Она вновь генерирует объект-исключение. Повторное возбуждение возможно только внутри составной инструкции, являющейся частью catch-обработчика:

```
catch ( exception eObj ) {
  if ( canHandle( eObj ) )
    // обработать исключение
    return;
  else
    // повторно возбудить исключение, чтобы его перехватил другой
    // catch-обработчик
```

```
        throw;
    }
```

При повторном возбуждении новый объект-исключение не создается. Это имеет значение, если catch-обработчик модифицирует объект, прежде чем возбудить исключение повторно.

```
    enum EHstate { noErr, zeroOp, negativeOp, severeError };

    void calculate( int op ) {
    try {
        // исключение, возбужденное mathFunc(), имеет значение
zeroOp
        mathFunc( op );
      }
      catch ( EHstate eObj ) {
        // что-то исправить

        // пытаемся модифицировать объект-исключение
        eObj = severeErr;

        // предполагалось, что повторно возбужденное исключение
будет
        // иметь значение severeErr
        throw;
      }
    }
```

Так как eObj не является ссылкой, то catch-обработчик получает копию объекта-исключения, так что любые модификации eObj относятся к локальной копии и не отражаются на исходном объекте-исключении, передаваемом при повторном возбуждении. Таким образом, переданный далее объект по-прежнему имеет тип zeroOp.

Чтобы модифицировать исходный объект-исключение, в объявлении исключения внутри catch-обработчика должна фигурировать ссылка:

```
    catch ( EHstate &eObj ) {
      // модифицируем объект-исключение
      eObj = severeErr;
```

```
    // повторно возбужденное исключение имеет значение severeErr
    throw;
  }
```

Теперь eObj ссылается на объект-исключение, созданный выражением throw, так что все изменения относятся непосредственно к исходному объекту. Поэтому при повторном возбуждении исключения далее передается модифицированный объект.

Таким образом, другая причина для объявления ссылки в catch-обработчике заключается в том, что сделанные внутри обработчика модификации объекта-исключения в таком случае будут видны при повторном возбуждении исключения.

7.6 Перехват всех исключений

Иногда функции нужно выполнить определенное действие до того, как она завершит обработку исключения, даже несмотря на то, что обработать его она не может. К примеру, функция захватила некоторый ресурс, скажем открыла файл или выделила память из хипа, и этот ресурс необходимо освободить перед выходом:

```
void manip() {
  resource res;
  res.lock();      // захват ресурса

  // использование ресурса
  // действие, в результате которого возбуждено исключение

  res.release();   // не выполняется, если возбуждено исключение
}
```

Если исключение возбуждено, то управление не попадет на инструкцию, где ресурс освобождается. Чтобы освободить ресурс, не пытаясь перехватить все возможные исключения, воспользуемся специальной конструкцией, позволяющей перехватывать любые исключения. Это не что иное, как предложение catch, в котором объявление исключения имеет вид (...) и куда управление попадает при любом исключении. Например:

```
// управление попадает сюда при любом возбужденном исключении
catch (...) {
  // здесь размещаем наш код
}
```

Конструкция catch(...) используется в сочетании с повторным возбуждением исключения. Захваченный ресурс освобождается внутри составной инструкции в catch-обработчике перед тем, как передать исключение по цепочке вложенных вызовов в результате повторного возбуждения:

```
void manip() {
  resource res;
  res.lock();
  try {
    // использование ресурса
    // действие, в результате которого возбуждено исключение
  }
  catch (...) {
    res.release();
    throw;
  }
  res.release();   // не выполняется, если возбуждено исключение
}
```

Чтобы гарантировать освобождение ресурса в случае, когда выход из manip() происходит в результате исключения, мы освобождаем его внутри catch(...) до того, как исключение будет передано дальше. Можно также управлять захватом и освобождением ресурса путем инкапсуляции в класс всей работы с ним. Тогда захват будет реализован в конструкторе, а освобождение – в автоматически вызываемом деструкторе.

Предложение catch(...) используется самостоятельно или в сочетании с другими catch-обработчиками. В последнем случае следует позаботиться о правильной организации обработчиков, ассоциированных с try-блоком.

Catch-обработчики исследуются по очереди, в том порядке, в котором они записаны. Как только найден подходящий, просмотр пре-

кращается. Следовательно, если предложение catch(...) употребляется вместе с другими catch-обработчиками, то оно должно быть последним в списке, иначе компилятор выдаст сообщение об ошибке:

```
try {
  stack.display();
  for ( int ix = 1; ix < 51; ++x )
  {
    // то же, что и выше
  }
}
catch ( pushOnFull ) { }
catch ( popOnEmpty ) { }
catch ( ... ) { }     // должно быть последним в списке catch-обработчиков
```

7.7 Спецификации исключений

По объявлениям методов pop() и push() класса iStack невозможно определить, что они возбуждают исключения. Можно, конечно, включить в объявление подходящий комментарий. Тогда описание интерфейса класса в заголовочном файле будет содержать документацию возбуждаемых исключений:

```
class iStack {
public:
  // ...

  void pop( int &value );   // возбуждает popOnEmpty
  void push( int value );   // возбуждает pushOnFull

private:
  // ...
};
```

Но такое решение несовершенно. Неизвестно, будет ли обновлена документация при выпуске следующих версий iStack. Кроме того, комментарий не дает компилятору достоверной информации о том, что никаких других исключений функция не возбуждает. Спецификация исключений позволяет перечислить в объявлении

функции все исключения, которые она может возбуждать. При этом гарантируется, что другие исключения функция возбуждать не будет.

Такая спецификация следует за списком формальных параметров функции. Она состоит из ключевого слова throw, за которым идет список типов исключений, заключенный в скобки. Например, объявления методов класса iStack можно модифицировать, добавив спецификации исключений:

```
class iStack {
public:
  // ...

  void pop( int &value ) throw(popOnEmpty);
  void push( int value ) throw(pushOnFull);

private:
  // ...
};
```

Гарантируется, что при обращении к pop() не будет возбуждено никаких исключений, кроме popOnEmpty, а при обращении к push() – только pushOnFull.

Объявление исключения – это часть интерфейса функции, оно должно быть задано при ее объявлении в заголовочном файле. Спецификация исключений – это своего рода "контракт" между функцией и остальной частью программы, гарантия того, что функция не будет возбуждать никаких исключений, кроме перечисленных.

Если в объявлении функции присутствует спецификация исключений, то при повторном объявлении этой же функции должны быть перечислены точно те же типы. Спецификации исключений в разных объявлениях одной и той же функции не суммируются:

```
// два объявления одной и той же функции
extern int foo( int parm = 0 ) throw(string);
// ошибка: опущена спецификация исключений
extern int foo( int parm ) { }
```

Исключения возбуждаются только при обнаружении определенных аномалий в поведении программы, и во время компиляции неиз-

вестно, встретится ли то или иное исключение во время выполнения. Поэтому нарушения спецификации исключений функции могут быть обнаружены только во время выполнения. Если функция возбуждает исключение, не указанное в спецификации, то вызывается unexpected() из стандартной библиотеки C++, а та по умолчанию вызывает terminate().

Необходимо уточнить, что unexpected() не вызывается только потому, что функция возбудила исключение, не указанное в ее спецификации. Все нормально, если она обработает это исключение самостоятельно, внутри функции. Например:

```
void recoup( int op1, int op2 ) throw(ExceptionType)
{
  try {
    // ...
    throw string("we're in control");
  }
  // обрабатывается возбужденное исключение
  catch ( string ) {
    // сделать все необходимое
  }
}  // все хорошо, unexpected() не вызывается
```

Функция recoup() возбуждает исключение типа string, несмотря на его отсутствие в спецификации. Поскольку это исключение обработано в теле функции, unexpected() не вызывается.

Нарушения спецификации исключений функции обнаруживаются только во время выполнения. Компилятор не сообщает об ошибке, если в выражении throw возбуждается исключение неуказанного типа. Если такое выражение никогда не выполнится или не возбудит исключения, нарушающего спецификацию, то программа будет работать, как и ожидалось, и нарушение никак не проявится:

```
extern void doit( int, int ) throw(string, exceptionType);
void action ( int op1, int op2 ) throw(string) {
  doit( op1, op2 );   // ошибки компиляции не будет
  // ...
}
```

Функция doit() может возбудить исключение типа exceptionType, которое не разрешено спецификацией action(). Однако функция компилируется успешно. Компилятор при этом генерирует код, гарантирующий, что при возбуждении исключения, нарушающего спецификацию, будет вызвана библиотечная функция unexpected().

Пустая спецификация показывает, что функция не возбуждает никаких исключений:

```
extern void no_problem () throw();
```

Если же в объявлении функции спецификация исключений отсутствует, то может быть возбуждено исключение любого типа.

Между типом возбужденного исключения и типом исключения, указанного в спецификации, не разрешается проводить никаких преобразований:

```
int convert( int parm ) throw(string)
{
  //...
  if ( somethingRather )
    // ошибка программы:
    // convert() не допускает исключения типа const char*
    throw "help!";
}
```

Выражение throw в функции convert() возбуждает исключение типа строки символов в стиле языка C. Созданный объект-исключение имеет тип const char*. Обычно выражение типа const char* можно привести к типу string. Однако спецификация не допускает преобразования типов, поэтому если convert() возбуждает такое исключение, то вызывается функция unexpected(). Для исправления ошибки выражение throw можно модифицировать так, чтобы оно явно преобразовывало значение выражения в тип string:

```
throw string( "help!" );
```

Спецификации исключений и указатели на функции

Спецификацию исключений можно задавать и при объявлении указателя на функцию. Например:

```
void (*pf)( int ) throw(string);
```

В этом объявлении говорится, что pf указывает на функцию, которая способна возбуждать только исключения типа string. Как и для объявлений функций, спецификации исключений в разных объявлениях одного и того же указателя не суммируются, они должны быть одинаковыми:

```
extern void (*pf) ( int ) throw(string);
// ошибка: отсутствует спецификация исключения
void (*pf)( int );
```

При работе с указателем на функцию со спецификацией исключений есть ограничения на тип указателя, используемого в качестве инициализатора или стоящего в правой части присваивания. Спецификации исключений обоих указателей не обязаны быть идентичными. Однако на указатель-инициализатор накладываются такие же или более строгие ограничения, чем на инициализируемый указатель (или тот, которому присваивается значение). Например:

```
void recoup( int, int ) throw(exceptionType);
void no_problem() throw();
void doit( int, int ) throw(string, exceptionType);

// правильно: ограничения, накладываемые на спецификации
// исключений recoup() и pf1, одинаковы
void (*pf1)( int, int ) throw(exceptionType) = &recoup;

// правильно: ограничения, накладываемые на спецификацию исключений no_problem(), более строгие,
// чем для pf2
void (*pf2)( ) throw(string) = &no_problem;

// ошибка: ограничения, накладываемые на спецификацию
// исключений doit(), менее строгие, чем для pf3
//
void (*pf3)( int, int ) throw(string) = &doit;
```

Третья инициализация не имеет смысла. Объявление указателя гарантирует, что pf3 адресует функцию, которая может возбуждать только исключения типа string. Но doit() возбуждает также исключения типа exceptionType. Поскольку она не подходит под ограничения, накладываемые спецификацией исключений pf3, то не может служить корректным инициализатором для pf3, так что компилятор выдает ошибку.

8. Построение псевдографического интерфейса

8.1 Основные положения

Библиотека ncurses предназначена для управления текстовым пользовательским интерфейсом (TUI) в терминале. Она обеспечивает удобный способ создания и обработки окон, цветов, атрибутов текста и других элементов текстового интерфейса в консольных приложениях.

Основные возможности библиотеки ncurses:

- Окна и панели: ncurses позволяет создавать и управлять окнами и панелями, которые могут быть размещены на экране терминала. Это облегчает разделение экрана на разные области и обновление контента в каждой области независимо друг от друга.
- Цвета и атрибуты: с помощью ncurses можно устанавливать различные атрибуты текста, такие как цвета, жирность, подчеркивание и прочее. Это позволяет создавать более яркий и информативный текстовый интерфейс.
- Управление клавишами: ncurses предоставляет функции для обработки клавиш, включая специальные клавиши (например, клавиши со стрелками, функциональные клавиши и т.д.). Это облегчает создание интерактивных приложений с поддержкой горячих клавиш и навигации.
- Управление курсором: с помощью ncurses можно управлять позицией курсора на экране, перемещать его, скрывать и отображать.
- Поддержка Unicode: ncurses поддерживает Unicode, что позволяет использовать символы разных языков и специальные символы для создания графических элементов в текстовом интерфейсе.

Рассмотрим простейший пример использования библиотеки ncurses.

Листинг 1. Инициализация и вывод текста на экран

```
#include <ncurses.h>

int main() {
   initscr(); // Инициализация ncurses
   raw(); // Включение режима сырых символов
   keypad(stdscr, TRUE); // Включение специальных клавиш
   noecho(); // Отключение отображения вводимых символов

   printw("Hello world!"); // Вывод текста на экран
   refresh(); // Обновление экрана

   getch(); // Ожидание нажатия клавиши
   endwin(); // Завершение работы с ncurses

   return 0;
}
```

В данном примере сначала осуществляется подключение заголовочного файла ncurses.h, в котором содержаться прототипы всех функций библиотеки. Затем в функции main производится инициализация библиотеки с помощью функции initscr. С помощью функции printw проиводится вывод текста на экран. Затем производится обновление содержимого экрана. После этого программа ожидает нажатия любой клавиши, завершает работы библиотеки и ,собственно, сама завершает свою работу.

В результате работы программы на экран будет выведено сообщение Hello world!, как показано на рисунке 8.1.

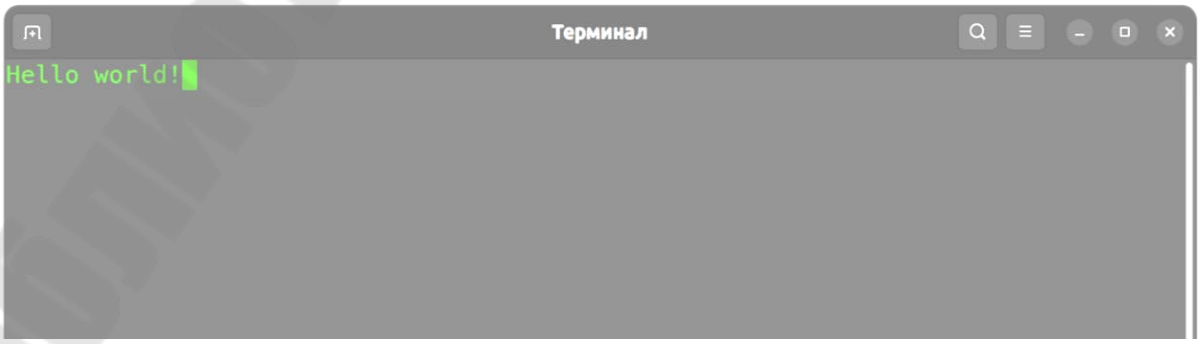

Рисунок 8.1 – Результат работы программы

Стоит отметить что данная библиотека не является встроенной для для компилятора GCC и при сборке необходимо указать флаг линковщика -lncurses для подключения данной библиотеки.

Команда для сборки в общем случае будет выглядеть так:

gcc example.c -lncurses -o example

где, example.c – имя файла с исходным кодом, а example – это имя результирующего исполняемого файла.

8.2 Основные функции библиотеки ncurses

Библиотека NCurses содержит множество функций для создания текстовых интерфейсов. Условно их можно разделить на следующие группы:

Инициализация и завершение

- Управление окнами.
- Ввод данных.
- Вывод данных.
- Управление атрибутами и цветами.
- Работа с панелями.

8.3 Управление окнами

Библиотека NCurses предоставляет возможность управлять окнами, которые могут быть использованы для организации интерфейса и разделения его на разные блоки. Основные функции для взаимодействия с окнами приведены в таблице 8.1.

Таблица 8.1 – Основные функции библиотеки для управления окнами

Функция	**Описание**
WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x)	Создает новое окно с указанными размерами (nlines, ncols) и начальными координатами (beginy, beginx). Возвращает указатель на созданное окно
int delwin(WINDOW *win)	Удаляет окно, на которое указывает win, освобождая все связанные с ним ресурсы. Возвращает OK при успешном удалении или ERR в случае ошибки

int wrefresh(WINDOW *win)	Обновляет содержимое окна win на экране. Возвращает OK при успешном обновлении или ERR в случае ошибки
int mvwin(WINDOW *win, int y, int x)	Перемещает курсор в окне win на новую позицию с координатами (y, x). Возвращает OK при успешном перемещении или ERR в случае ошибки
int wmove(WINDOW *win, int y, int x)	Перемещает курсор в окне win на новую позицию с координатами (y, x). Возвращает OK при успешном перемещении или ERR в случае ошибки
int box(WINDOW *win, chtype verch, chtype horch)	Рисует рамку вокруг окна win с вертикальными линиями, обозначенными символом verch, и горизонтальными линиями, обозначенными символом horch. Возвращает OK при успешном создании рамки или ERR в случае ошибки

Рассмотрим пример, в котором создается окно с рамкой. Фон окна синий, рамка – белая. В данном окне выводится текст « Test Window» красным цветом.

Листинг 2. Пример создания окна

```
#include <iostream>
#include <ncurses.h>
#include <string.h>

void update_screen(WINDOW *win, const char *text) {
    int height, width;
    getmaxyx(win, height, width);
    // Устанавливаем синий фон и рисуем рамку
    wbkgd(win, COLOR_PAIR(1));
    wattron(win, COLOR_PAIR(1));
    box(win, 0, 0);
    wattroff(win, COLOR_PAIR(1));

    // Выводим текст "Test Window" по центру окна красным цветом
    int text_y = height / 2;
    int text_x = (width - strlen(text)) / 2;
    wattron(win, COLOR_PAIR(2)); // Включение красного цвета текста
    mvwprintw(win, text_y, text_x, "%s", text);
```

```
    wattroff(win, COLOR_PAIR(2)); // Выключение красного цвета текста

    // Обновление окна на экране
    wrefresh(win);
}

int main() {
    initscr(); // Инициализация библиотеки NCurses
    start_color(); // Инициализация цветовой поддержки

    // Определение цветовых пар
    init_pair(1, COLOR_WHITE, COLOR_BLUE); // Белый текст на синем фоне для рамки
    init_pair(2, COLOR_RED, COLOR_BLUE);    // Красный текст на синем фоне для надписи

    int height = 10;
    int width = 30;
    int start_y = (LINES - height) / 2;
    int start_x = (COLS - width) / 2;

    // Создание окна
    WINDOW *win = newwin(height, width, start_y, start_x);

    // Настраиваем экран для немедленного вывода
    timeout(0);
    curs_set(0);
    const char *text = "Test Window";

    // Цикл обработки событий клавиатуры
    int ch = 0;
    while (ch != 'q') {
        update_screen(win, text);
        ch = getch();
    }

    // Удаление окна и завершение работы с NCurses
    delwin(win);
```

```
    endwin();

    return 0;
}
```

Функция update_screen осуществляет обновление экрана, при различных действия в приложении. Для работы данной функции передается указатель на текущее окно WINDOW *win, const и текст который необходимо в него вывести char *text. В начале функции определяется размер нашего окна с помощью команды getmaxyx. Так как интерфейс основан на символах псевдографики, то размерами окна является количество строк и столбцов. Эти ззначения заносятся в переменные height и width соответственно. После этого непосредственно рисуется окно: сначала задается синий фон на заданной области с помощью функции wbkgd, а затем рисуется рамка белого цвета. После этого осуществляется вывод текста Test Window красным цветом. Затем, с помощью функции wrefresh, произвдится обновление окна на экране.

В основной функции данной программы сначала производится инициализация библиотеки и цветовой поддержки. Затем производится определение цветовых пар. Создается окно и настраивается экран для немедленного вывода. После формирования текстовой строки, которую необходимо вывести в окне, запускается цикл, в котором производится обновление окна пока не будет введен символ q. Если данный символ введен, то программа завершает все действия и производит выход.

Функция mvwin() из библиотеки ncurses в C и C++ используется для перемещения окна на экране. Ее прототип выглядит следующим образом:

```
int mvwin(WINDOW *win, int y, int x);
```

Данная функция принимает три аргумента:

- win: Указатель на окно, которое нужно переместить. Это окно должно быть создано заранее с помощью функции newwin().
- y: Новая координата y (вертикальная) окна. Она указывает на новую верхнюю строку окна относительно экрана.
- x: Новая координата x (горизонтальная) окна. Это указывает на новую левую колонку окна относительно экрана.

Она возвращает OK (обычно это 0) в случае успешного выполнения и ERR (обычно это -1) в случае ошибки. Стоит обратить внимание, что mvwin() просто перемещает окно, но не обновляет экран. Чтобы увидеть перемещенное окно, вам нужно вызвать wrefresh() или аналогичную функцию. Также стоит отметить, что функция mvwin() перемещает только окно, но не очищает его предыдущее положение. Если вам нужно "стереть" старое положение окна, вам придется сделать это вручную.

Пример использования функции mvwin():

```
WINDOW *win = newwin(10, 10, 0, 0);  // Создаем новое окно размером 10x10 в позиции (0,0)
mvwin(win, 5, 5);  // Перемещаем окно в позицию (5,5)
wrefresh(win);  // Обновляем экран, чтобы увидеть перемещенное окно
```

Рассмотри программу, которая с помощью функции mvwin через заданные промежутки времени перемещает окно с надписью move window в случайное местоположение.

```
#include <ncurses.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

#define DELAY 2000000
void erase_window(WINDOW* win) {
   int y, x, max_y = 0, max_x = 0;


   // Получаем текущее положение окна
   getbegyx(win, y, x);

   // Получаем размеры окна
   getmaxyx(win, max_y, max_x);

   // Создаем новое окно в том же месте и размере, что и старое
   WINDOW *erase_win = newwin(max_y, max_x, y, x);
```

```
    // Заполняем окно цветом фона
    wbkgd(erase_win, COLOR_PAIR(1));

    // Обновляем и удаляем окно
    wrefresh(erase_win);
    delwin(erase_win);
}

int main() {
    // Инициализация ncurses
    initscr();
    start_color();
    noecho();
    curs_set(FALSE);

    // Определение цветов
    init_pair(1, COLOR_WHITE, COLOR_BLUE); // белый на синем
    init_pair(2, COLOR_RED, COLOR_BLUE); // красный на синем

    // Создание нового окна
    WINDOW *win = newwin(5, 20, 0, 0);
    keypad(win, TRUE);

    // Установка цвета рамки
    wattron(win, COLOR_PAIR(1));
    box(win, 0, 0);

    // Установка цвета надписи
    wattron(win, COLOR_PAIR(2));
    mvwprintw(win, 2, 2, "move window");
    wattroff(win, COLOR_PAIR(2));
    // Обновление окна
    wrefresh(win);

    // Случайное перемещение окна
    srand(time(NULL));
    int max_y = LINES - 5, max_x = COLS - 20;
    for(int i = 0; i < 10; ++i) {
        erase_window(win);
```

```
        int new_y = rand() % max_y;
        int new_x = rand() % max_x;
        mvwin(win, new_y, new_x);

        wrefresh(win);
        usleep(DELAY); // задержка в 2 секунды
    }

    // Завершение работы с ncurses
    delwin(win);
    endwin();
    return 0;
}
```

Программа начинается с инициализации библиотеки ncurses с помощью функции initscr(). Эта функция также очищает экран и устанавливает режимы ввода и вывода. Затем функция start_color() инициализирует цвета, и noecho() отключает отображение вводимых пользователем символов на экране. Функция curs_set(FALSE) скрывает курсор. Цветовые пары для рамки окна и текста внутри окна устанавливаются с помощью функции init_pair(). Далее создается новое окно с помощью функции newwin(). Эта функция принимает высоту и ширину окна, а также координаты начала окна. С помощью функции wattron(), устанавливается цвет рамки окна и текста внутри окна. Текст "move window" выводится в окне с помощью функции mvwprintw(). Окно обновляется с помощью функции wrefresh(), чтобы отобразить рамку и текст. Затем начинается цикл, в котором окно перемещается в случайные позиции на экране. Это достигается с помощью функции mvwin(), которая перемещает окно в новые координаты. После каждого перемещения окно обновляется с помощью wrefresh(), а затем программа ожидает 2 секунды с помощью функции usleep(). Перед каждым перемещением окна вызывается функция erase_window(). Эта функция создает новое окно в том же месте и размере, что и старое, заполняет его цветом фона и затем удаляет его, тем самым стирая старое окно. После завершения цикла, окно удаляется с помощью функции delwin(), и работа с ncurses завершается с помощью функции endwin().

8.4 Ввод и вывод данных

Для ввода данных в библиотеке ncurses существует ряд функций. Перечень этих функций находится в таблице 8.2. Также в таблице 8.3 находится перечень функций для вывода данных.

Таблица 8.2 – Функции библиотеки для ввода даных

Функция	**Описание**
int getch(void);.	Эта функция ожидает и возвращает символ, введенный пользователем
int wgetch(WINDOW *win);	Эта функция аналогична getch(), но она читает символ из указанного окна
int mvgetch(int y, int x);	Эта функция перемещает курсор в указанную позицию, а затем ожидает и возвращает символ, введенный пользователем
int mvwgetch(WINDOW *win, int y, int x);	Эта функция аналогична mvgetch(), но она читает символ из указанного окна
char *getstr(char *str);	Эта функция читает строку, введенную пользователем, и сохраняет ее в указанном массиве символов
char *wgetstr(WINDOW *win, char *str);	Эта функция аналогична getstr(), но она читает строку из указанного окна
char *mvgetstr(int y, int x, char *str);	Эта функция перемещает курсор в указанную позицию, а затем читает строку, введенную пользователем, и сохраняет ее в указанном массиве символов
char *mvwgetstr(WINDOW *win, int y, int x, char *str);	Эта функция аналогична mvgetstr(), но она читает строку из указанного окна.

Таблица 8.3 – Функции библиотеки для вывода даных

Функция	Описание
printw(const char *fmt, ...):	Эта функция используется для вывода форматированной строки в текущую позицию курсора в стандартном окне. Это аналог функции printf из стандартной библиотеки C
mvprintw(int y, int x, const char *fmt, ...):	Эта функция сначала перемещает курсор в позицию (y, x), а затем выводит форматированную строку. Это аналог printw, но с предварительным перемещением курсора
wprintw(WINDOW *win, const char *fmt, ...):	Эта функция аналогична printw, но позволяет выводить строку в указанное окно win
mvwprintw(WINDOW *win, int y, int x, const char *fmt, ...):	Эта функция аналогична mvprintw, но позволяет выводить строку в указанное окно win с предварительным перемещением курсора
addch(chtype ch):	Эта функция добавляет символ ch в текущую позицию курсора в стандартном окне и перемещает курсор на одну позицию вправо
mvaddch(int y, int x, chtype ch):	Эта функция перемещает курсор в позицию (y, x), а затем добавляет символ ch
waddch(WINDOW *win, chtype ch):	Эта функция добавляет символ ch в текущую позицию курсора в указанном окне win
mvwaddch(WINDOW *win, int y, int x, chtype ch):	Эта функция перемещает курсор в позицию (y, x) в окне win, а затем добавляет символ ch

Все эти функции блокируют выполнение программы и ожидают ввода от пользователя, если не установлен неблокирующий режим. Важно отметить, что функции для чтения строк (getstr(), wgetstr(), mvgetstr(), mvwgetstr()) не проверяют переполнение буфера, поэтому их использование может привести к проблемам безопасности. Вместо них лучше использовать функции getnstr(), wgetnstr(), mvgetnstr(), mvwgetnstr(), которые принимают дополнительный аргумент для указания максимального числа символов для чтения.

Рассмотри простой пример программы, которая с помощью функции getch считывает символ с клавиатуры, выводит его на экран и завершает свою работу.

```
#include <ncurses.h>

int main() {
    // Инициализация ncurses
    initscr();
    noecho();
    curs_set(FALSE);

    // Считывание символа с клавиатуры
    printw("Press any key: ");
    int ch = getch();

    // Вывод символа на экран
    printw("\nInput char: %c", ch);

    // Ожидание нажатия клавиши перед завершением программы
    getch();

    // Завершение работы с ncurses
    endwin();
    return 0;
}
```

В данной программе происходит инициализация библиотеки с помощью функции initscr. Затем с помощью функции noecho отключается вывод в терминал нажатых символов на клавиатуре. С помощью функции curs_set отключается видимость курсора. После этого выводится надпись "Press any key: " и с помощью функции getch ожидается ввод символа. После того, как символ введен, программа выводит его на экран и завершает свою работу.

Теперь рассмотрим более сложный пример, но и более приближенный к реальным задачам. В программе, приведенной ниже, создается овно, для ввода двух чисел A и B. После ввода обоих числел происходит их сложение и результат выводится на экран.

```
#include <ncurses.h>
#include <string>
```

```
int main() {
    initscr(); // Инициализация ncurses
    cbreak(); // Отключение буферизации ввода
    echo(); // Включение отображения введенных символов

    int height = 5;
    int width = 30;
    int start_y = (LINES - height) / 2;
    int start_x = (COLS - width) / 2;

    WINDOW *win = newwin(height, width, start_y, start_x); // Создание нового окна
    box(win, 0, 0); // Отрисовка рамки окна
    mvwprintw(win, 1, 1, "Enter A: ");
    mvwprintw(win, 2, 1, "Enter B: ");
    wrefresh(win); // Обновление окна

    char str_a[10], str_b[10];
    mvwgetnstr(win, 1, 9, str_a, 10); // Запрос на ввод числа A
    mvwgetnstr(win, 2, 9, str_b, 10); // Запрос на ввод числа B

    // Конвертирование строк в числа
    int a = std::stoi(str_a);
    int b = std::stoi(str_b);

    wclear(win); // Очистка окна
    wrefresh(win); // Обновление окна
    delwin(win); // Удаление окна

    printw("The sum of A and B is: %d", a+b); // Вывод результата
    refresh(); // Обновление экрана

    getch(); // Ждем нажатия клавиши
    endwin(); // Завершение работы с ncurses

    return 0;
}
```

Сначала программа инициализирует библиотеку ncurses с помощью initscr(), отключает буферизацию ввода с помощью cbreak()

и включает отображение введенных символов с помощью echo(). Затем программа создает новое окно с помощью newwin(), указывая высоту, ширину и координаты начала окна. Параметры высоты и ширины окна устанавливаются в 5 и 30 соответственно, а начальные координаты окна вычисляются так, чтобы окно было отцентровано на экране. После создания окна, программа рисует рамку вокруг него с помощью функции box() и выводит две строки текста внутри окна с помощью mvwprintw(). Эти строки текста просят пользователя ввести два числа - "A" и "B". Затем окно обновляется с помощью wrefresh() чтобы отобразить внесенные изменения. Затем программа ожидает ввод от пользователя для чисел "A" и "B" с помощью mvwgetnstr(), сохраняя введенные значения как строки в переменных str_a и str_b соответственно. Эти строки затем конвертируются в числа с помощью std::stoi(), и результаты сохраняются в переменных a и b.

После получения и конвертации введенных значений, программа очищает и удаляет окно с помощью wclear(), wrefresh() и delwin(). И вычисляет сумму введенных чисел, после чего выводит результат на экран с помощью printw(). Программа затем ожидает нажатия любой клавиши пользователем с помощью getch() перед завершением работы и деинициализацией ncurses с помощью endwin(). Это позволяет пользователю увидеть результат перед тем, как программа закончит работу.

В следуюшем примере реализован ввд текстовых данных. Программа просить пользователя ввести фамилию, имя и отчество студента, а так же его группу. После ввода данных, окно удаляется, а данные выводятся на экран.

```
#include <ncurses.h>
#include <string>

struct Student {
    std::string lastName;
    std::string firstName;
    std::string middleName;
    std::string group;
};

int main() {
    initscr();
    start_color();
```

```
init_pair(1, COLOR_WHITE, COLOR_BLUE);

// Получаем размеры экрана
int maxY, maxX;
getmaxyx(stdscr, maxY, maxX);

// Вычисляем позицию окна, чтобы разместить его по центру
int winY = (maxY - 10) / 2;
int winX = (maxX - 40) / 2;

WINDOW* win = newwin(10, 40, winY, winX);
wbkgd(win, COLOR_PAIR(1));
box(win, 0, 0);
wrefresh(win);

Student student;
char lastName[50];
char firstName[50];
char middleName[50];
char group[50];

mvwprintw(win, 1, 1, "LastName: ");
mvwprintw(win, 2, 1, "FirstName: ");
mvwprintw(win, 3, 1, "MiddleName: ");
mvwprintw(win, 4, 1, "Group: ");
wrefresh(win);

echo();
mvwgetnstr(win, 1, 12, lastName, 50);
mvwgetnstr(win, 2, 13, firstName, 50);
mvwgetnstr(win, 3, 14, middleName, 50);
mvwgetnstr(win, 4, 8, group, 50);

student.lastName = lastName;
student.firstName = firstName;
student.middleName = middleName;
student.group = group;
```

```
    endwin();
    printf("LastName: %s\n", student.lastName.c_str());
    printf("FirstName: %s\n", student.firstName.c_str());
    printf("MiddleName: %s\n", student.middleName.c_str());
    printf("Group: %s\n", student.group.c_str());
    printf("Press any key to exit...\n");
    getchar();

    return 0;
}
```

Сначала программа инициализирует библиотеку ncurses с помощью функции initscr(). Это позволяет использовать функции ncurses для управления выводом на терминал. Затем программа включает поддержку цветов с помощью функции start_color(). Создается новая цветовая пара (белый текст на синем фоне) с помощью функции init_pair(). Программа получает размеры текущего терминала с помощью функции getmaxyx(). Эти размеры затем используются для вычисления координат, необходимых для размещения нового окна по центру терминала. Создается новое окно с помощью функции newwin(), которое размещается по центру экрана. Фон нового окна устанавливается в синий цвет с помощью функции wbkgd(). Рисуется рамка вокруг окна с помощью функции box(). Экран обновляется с помощью функции wrefresh() для отображения нового окна. Программа выводит подсказки для ввода данных в окне. Включается отображение введенного текста с помощью функции echo(). Программа ожидает ввода данных от пользователя с помощью функции mvwgetnstr() для каждого поля. Введенные данные сохраняются в структуре Student. Работа с ncurses завершается с помощью функции endwin(). Программа выводит сохраненные данные на экран. Затем ожидает нажатия любой клавиши, после чего завершает свою работу.

Результат работы программы изображен на рисунке 8.2.

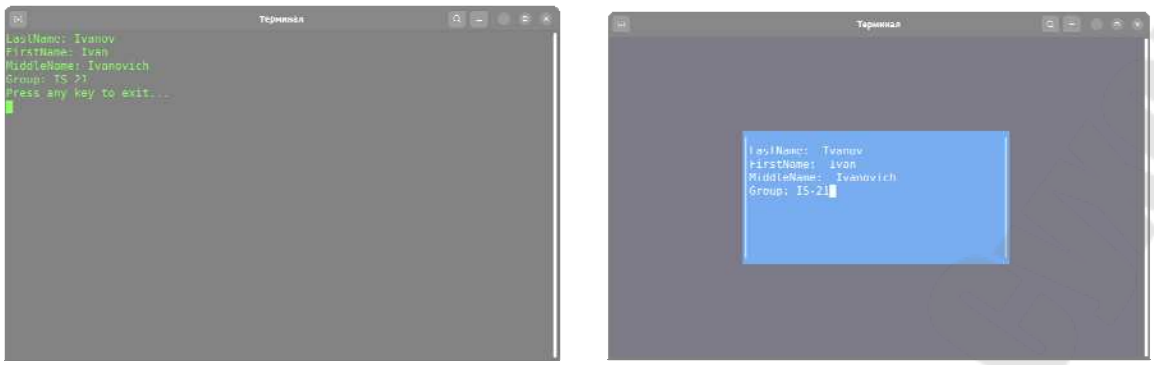

Рисунок 8.2 – Результат работы программы по вводу текстовых данных

8.5 Управление атрибутами и цветами

Библиотека ncurses предоставляет множество функций для управления атрибутами и цветами текста. Перечень данных функций приведен в таблице 4.

Таблица 8.4 – Фнкции управления атрибутами и цветами

Функция	**Описание**
int startcolor(void);	Инициализирует цветовую подсистему ncurses.
int initpair(short pair, short f, short b);`	нициализирует цветовую пару. `pair` – номер пары, `f` – номер цвета переднего плана, `b` – номер цвета заднего плана
#define COLOR_PAIR(n)	Макрос, который возвращает атрибут, который можно использовать для цвета текста. n - номер цветовой пары
int attron(int attrs);	Включает атрибуты, указанные в attrs. Атрибуты могут быть значением COLORPAIR(n) или другими атрибутами, вроде ABOLD
int attroff(int attrs);	Выключает атрибуты, указанные в attrs
int attrset(int attrs);	Устанавливает (включает) только атрибуты, указанные в attrs, остальные атрибуты выключаются
int standend(void);	Выключает все атрибуты
int standout(void);	Включает выделение (обычно обратные цвета)
int initcolor(short color, short r, short g, short b);	Изменяет определение цвета. color - номер цвета, r, g, b - значения красного, зеленого и синего соответственно

bool hascolors(void);`	Возвращает TRUE, если терминал поддерживает цвета, и FALSE в противном случае
bool canchangecolor(void);`	Возвращает TRUE, если терминал поддерживает изменение цветов, и FALSE в противном случае
#define COLORS`	Макрос, который содержит число определенных цветов
#define COLOR_PAIRS	Макрос, который содержит число доступных цветовых пар

Перечисленные в таблице 4 функции и макросы представляют основные возможности управления атрибутами и цветами в библиотеке ncurses. Они могут варьироваться в зависимости от версии и конкретного производителя. Всегда следует обращаться к документации для получения наиболее точной информации.

Рассмотрим пример, в котором реализован вывод текста с различными атрибутами.

```
#include <ncurses.h>

int main()
{
   // Инициализация ncurses
   initscr();

   // Проверка поддержки цветов
   if (has_colors() == FALSE) {
      endwin();
      printf("Your terminal does not support color\n");
      return 1;
   }

   // Проверка возможности изменения цветов
   if (can_change_color() == TRUE) {
      // Задаем новый цвет
      init_color(COLOR_RED, 700, 0, 0); // Более яркий красный цвет
   }
```

```
    // Инициализация цветов
    start_color();

    // Определение пары цветов
    init_pair(1, COLOR_RED, COLOR_BLACK);

    // Включение атрибутов
    attron(COLOR_PAIR(1));
    attron(A_BOLD);

    // Вывод текста
    printw("Bold Red Text\n");

    // Выключение атрибутов
    attroff(COLOR_PAIR(1));
    attroff(A_BOLD);

    // Включение выделения
    standout();
    printw("Standout Text\n");
    standend();
    // Вывод текста
    printw("Normal Text\n");

    // Обновление экрана
    refresh();

    // Ожидание ввода
    getch();

    // Завершение работы с ncurses
    endwin();

    return 0;
}
```

Сначала программа инициализирует библиотеку ncurses с помощью функции initscr(). Это необходимо для использования функций ncurses. Далее программа проверяет, поддерживает ли терминал цвета с помощью функции has_colors(). Если цвета не поддерживаются,

программа завершает работу с ncurses и выводит сообщение об ошибке. Если цвета поддерживаются, программа проверяет, можно ли изменить цвета с помощью функции can_change_color(). Если это возможно, программа задает новый красный цвет с помощью функции init_color(). Затем программа инициализирует цветовую подсистему с помощью функции start_color(). Программа определяет новую пару цветов с помощью функции init_pair(). Эта пара состоит из красного цвета для текста и черного цвета для фона. Далее программа включает атрибуты этой пары цветов и жирного шрифта с помощью функции attron() и выводит текст "Bold Red Text". Затем программа выключает эти атрибуты с помощью функции attroff(). Программа включает выделение текста с помощью функции standout(), выводит текст "Standout Text" и выключает выделение с помощью функции standend(). Далее программа выводит текст "Normal Text" без каких-либо атрибутов. Программа обновляет экран с помощью функции refresh() для отображения всего выведенного текста. Программа ожидает ввода с помощью функции getch(), чтобы пользователь мог увидеть результаты перед завершением программы. Наконец, программа завершает работу с ncurses с помощью функции endwin(). На рисунке 8.3 представлен результат работы программы.

Рисунок 8.3 – Фрмирование текста с различными атрибутами

8.6 Работа с панелями

Библиотека ncurses для C++ предоставляет функциональность для работы с панелями. Панели представляют собой слои, которые можно наложить друг на друга в окне вашего терминала. Это позволяет создавать более сложные пользовательские интерфейсы в среде командной строки, так как вы можете управлять, какие панели видимы, их порядком и содержимым. Перечень функций для работы с панелями приведен в таблице 8.5.

Таблица 8.5 – Функции для работы с панелями

Функция	**Описание**
PANEL* new_panel(WINDOW *win):	Ассоциирует новую панель с изначально невидимым окном. Возвращает указатель на новую панель
int del_panel(PANEL *pan):	Удаляет панель из стека, но не удаляет окно, связанное с панелем. Возвращает OK в случае успеха и ERR в случае ошибки
int hide_panel(PANEL *pan):	Скрывает панель (делает ее невидимой). Возвращает OK в случае успеха и ERR в случае ошибки
int show_panel(PANEL *pan):	Делает панель видимой. Если панель уже была видимой, то она помещается поверх остальных панелей. Возвращает OK в случае успеха и ERR в случае ошибки
int top_panel(PANEL *pan):	Помещает панель на вершину стека. Возвращает OK в случае успеха и ERR в случае ошибки
int bottom_panel(PANEL *pan):	Помещает панель на дно стека. Возвращает OK в случае успеха и ERR в случае ошибки
PANEL* panel_above(const PANEL *pan):	Возвращает панель, находящуюся выше указанной панели в стеке. Если передан NULL, то возвращает нижнюю панель. Если достигнута вершина стека, возвращает NULL
PANEL* panel_below(const PANEL *pan):	Возвращает панель, находящуюся ниже указанной панели в стеке. Если передан NULL, то возвращает верхнюю панель. Если достигнуто дно стека, возвращает NULL.

Окончание таблицы 8.5

Функция	Описание
int set_panel_userptr(PANEL *pan, void *uptr):	Устанавливает указатель пользователя для указанной панели. Возвращает OK в случае успеха и ERR в случае ошибки. void* panel_userptr(const PANEL *pan): Возвращает указатель пользователя для указанной панели
WINDOW* panel_window(const PANEL *pan):	Возвращает окно, ассоциированное с указанной панелью
void update_panels(void):	Обновляет виртуальную копию экрана, синхронизируя ее с текущим состоянием стека панелей
void move_panel(PANEL *pan, int starty, int startx):	Перемещает панель и ассоциированное с ней окно в новую позицию

Рассмотрим простой пример программы, которая создает две панели и выводит на них текст.

```
#include <ncurses.h>
#include <panel.h>

int main()
{
   /* Инициализация ncurses */
   initscr();
   cbreak();
   noecho();
   keypad(stdscr, TRUE);

   /* Создание двух окон для панелей */
   WINDOW *my_wins[2];
   my_wins[0] = newwin(10, 20, 1, 10);  // Создание нового окна
   box(my_wins[0], 0, 0);               // Нанесение рамки на окно
   mvwprintw(my_wins[0], 5, 5, "Panel 1");      // Добавление надписи
"Panel 1"
```

```
    my_wins[1] = newwin(10, 20, 1, 40);  // Создание другого нового окна
    box(my_wins[1], 0, 0);               // Нанесение рамки на окно
    mvwprintw(my_wins[1], 5, 5, "Panel 2");      // Добавление надписи "Panel 2"

    /* Создание двух панелей */
    PANEL *my_panels[2];
    my_panels[0] = new_panel(my_wins[0]);  // Привязка первого окна к панели
    my_panels[1] = new_panel(my_wins[1]);  // Привязка второго окна к панели

    /* Обновление стека панелей и вывод на экран */
    update_panels();
    doupdate();

    /* Ожидание нажатия клавиши пользователем */
    getch();


    /* Удаление панелей и окон, завершение работы с ncurses */
    del_panel(my_panels[0]);
    del_panel(my_panels[1]);
    delwin(my_wins[0]);
    delwin(my_wins[1]);
    endwin();

    return 0;
}
```

Программа начинается с инициализации библиотеки ncurses вызовом функции initscr(). Это создает виртуальное окно, которое соответствует размеру терминала. Затем, функции cbreak(), noecho(), и keypad() используются для установки режимов ввода. cbreak() позволяет программе немедленно принимать ввод пользователя, noecho() предотвращает отображение введенных пользователем символов на экране, а keypad() позволяет использовать специальные клавиши, такие как стрелки. Далее, программа создает два окна с помощью функции newwin(). Оба окна имеют размер 10x20 и расположены на одной

высоте (1 строка от верха экрана), но в разных столбцах (10 и 40 соответственно). Функция box() используется для добавления рамки вокруг каждого окна. Функция mvwprintw() используется для вывода надписей "Panel 1" и "Panel 2" в центре каждого окна. Затем, каждое окно привязывается к панели с помощью функции new_panel(). Функция update_panels() вызывается для обновления стека панелей, чтобы учесть все изменения. Функция doupdate() обновляет физический экран, чтобы он совпадал с виртуальным экраном. Далее, программа ожидает нажатия клавиши пользователем с помощью getch(). После нажатия клавиши, панели и окна удаляются с помощью функций del_panel() и delwin(), а затем функция endwin() завершает работу с ncurses и возвращает терминал в нормальный режим.

Таким образом, данная программа демонстрирует базовое использование панелей в библиотеке ncurses, позволяя создавать сложные пользовательские интерфейсы в командной строке.

9. Потоковый ввод-вывод

Библиотека потокового ввода-вывода iostream может быть использована в качестве альтернативы известной стандартной библиотеки ввода-вывода языка С – stdio. Для нас библиотека iostream интересна как прекрасный пример объектно-ориентированного проектирования так как содержит многие характерные приемы и конструкции. В основе ООП подхода, реализуемого средствами iostream, лежит предположение о том, что объекты обладают знанием того, какие действия следует предпри-нимать при вводе и выводе.

При пользовании библиотекой iostream ошибки, связанные с "перепутыванием" типов данных, исключены. Если вы используете в операции ввода-вывода переменную типа unsigned long, то вызывается подпрограмма, ответственная именно за этот тип.

Библиотека stdio поддерживает средства языка С, позволяющее использовать переменное число параметров. Но такая гибкость дается не даром – на этапе компиляции проверка соответствия между спецификацией формата, как в функциях printf() и scanf(), не выполняется.

В библиотеке iostream применен другой подход. Операторы ввода-вывода оформляются в виде выражений с применением переопределяемых функций-операторов для каждого из типов данных, встречающихся в выражении. Если вам необходимо использовать новый тип данных в операциях ввода-вывода, вы можете расширить библиотеку iostream своими функциями-операторами.

Библиотека iostream более медленная, чем stdio, но это небольшая плата за надежность и расширяемость, базирующиеся на возможностях объектно-ориентированных средств вывода.

9.1 Простое внесение

В данном примере реализовано две программы, первая с использованием библиотеки stdio.h а вторая с использованием iostream.h.

Вариант на стандартном С	**Вариант на С++**
#include <stdio.h> int i; char buff[80]; printf ("Введите число:"); scanf("%d", &i); printf ("Введите символьную строку:"); gets (buff); printf ("Вы ввели число: %d\n Вы ввели строку: %s\n", i, buff);	#include <iostream.h> int i; char buff[80]; cout << "Введите число и символьную строку:"; cin >> i >> buff; cout << "Вы ввели число:" << i << "\n" << "Вы ввели строку:" << buff << "\n";

Сообщения, выводимые программой, будут иметь вид:

Вы ввели число: 12

Вы ввели строку: My string

В первом варианте программы для вывода информации на экран применена функция printf(). В качестве параметра ей передается строка форматирования и переменные, значения которых необходимо вывести. Для того чтобы вывести числовое значение применяется спецификатор %d, а для вывода строки применяется спецификатор %s. Как видно из примера спецификатор в строке форматирования должен строго совпадать с типом данных, иначе произойдет ошибка или результат вывода на экран будет не таким как ожидается.

Во второй программе для вывода такой же информации применяется библиотека iostream. Данная программа выводит абсолютно то же самое, но отличается во всем остальном. Вывод в ней осуществляется в результате выполнения выражения, а не вызова отдельной функции типа printf(). Приведенное выражение называется выражением внесения, так как текст в данном случае вносится в выходной поток.

Для того чтобы понять как работает данная технология необходимо более подробно разобраться как компилятор интепритирует данную комманду. Первым шагом к пониманию операции внесения будет расстановка скобок в выражении так, чтобы можно было понять, каким образом оно интерпретируется компилятором:

(((((cout<<"Вы ввели число:")<<i)<<"\n Вы ввели строку:")<< buff)<<"\n");

Объект cout – это предопределенный объект класса iostream, который используется для вывода. Корме него существуют еще

cin – стандартный поток ввода

cerr – поток для вывода сообщений об ошибках.

Когда компилятор разбирает приведенное выше выражение, он начинает с самого высокого уровня вложенности скобочной структуры и на этом уровне находит:

cout<<"Вы ввели число:"

В процессе анализа этого выражения компилятор пытается отыскать функцию – operator<<(), имеющую в качестве левого операнда объект класса ostream, а в качестве правого – целое. Описание переопределяемой функции operator << (ostream &, char*) содержится в заголовочном файле iostream.h. Здесь компилятор C++ преобразует исходное выражение в более пригодное для дальнейшей обработки. В результате получается следующее:

operator<<(cout, "Вы ввели число:")

Когда функция operator << (ostream &, char*) будет выполнена, она выведет свой аргумент (строку) и примет значение объекта cout (ее первый аргумент). "Вы ввели число:" выведется на экран, и подвыражением, находящимся на самом глубоком уровне вложенности, станет

cout << i

Теперь компилятор ищет функцию operator <<(ostream &, int). Объект iostream содержит функции операторы для всех встроенных типов. Процесс продолжается до тех пор, пока выражение внесения не будет сведено к набору вызовов функции operatop << ().

В итоге, выполнение этой сточки приведет к последовательному вызову нескольких функций–операторов (таблица 9.1).

Таблица 9.1 – Функции операторов

Функции	Левый операнд	Правый операнд	Возвращаемое значение
operator (ostream &, char*)	cout	"Вы ввели число:"	cout
operator (ostream &, int)	cout	i	cout
operator (ostream &, char*)	cout	"\n Вы ввели строку:"	cout
operator (ostream &, char*)	cout	buff	cout
operator (ostream &, char*)	cout	"\n"	cout

Библиотека iostream чрезвычайно устойчива к ошибкам, связанным с неправильным употреблением типов переменных, так как каждый тип при вводе-выводе обслуживается свой собственной функцией извлечения и внесения.

9.2 Выражение извлечения

Как и внесение, извлечение выполняется в C++ переопределяемыми функциями–операторами, обращения к которым подставляются компилятором в зависимости от типов данных, используемых в выражении. Рассмотри еще раз предыдкщий пример.

Выражение извлечения в данной программе – это выражение, которое использует cin, предопределенный объект оператора iostream.

cin >> i >> buff;

При анализе этой конструкции компилятор ведет себя так же, как и при анализе выражения внесения, подставляя вызовы функции в соответствии с типами используемых переменных.

Если на вход поступит "строка 12", то программа будет в большом затруднении при попытке интерпретировать "строка" как число. Однако библиотека iostream, в отличие от scanf(), производит контроль ошибок после ввода каждого значения. Кроме того, iostream может быть расширена введением операторов для новых типов.

Одна из наиболее распространенных ошибок при использовании scanf() – это задание вместо адресов аргументов их значений. Другая распространенная ошибка – путаница в использовании модификаторов форматов. При работе с iostream такого не бывает, так как проверка соответствия типов – неотъемлемая часть процесса ввода-вывода.

Компилятор обеспечивает вызов функций-операторов, строго соответствующих используемым типам.

9.3 Создание собственных функции внесения и извлечения

Операторы >> и << можно перегружать, причем, можно создавать собственные операторы изалечения и внесения для собственных типов данных. В общем виде, операция внесения имеет следующую форму:

```
ostream& operator << (ostream& stream, имя_класса& obj) {
        stream << ...       // вывод элементов объекта obj
        // в поток stream, используя готовые функции внесения
return stream;
}
```

Аналогичным образом может быть определена функция извлечения:

```
istream& operator >> (istream& stream, имя_класса& obj)
{
        stream >> ...       // ввод элементов объекта obj
        // из потока stream, используя готовые функции внесения
return stream;
}
```

Функции внесения и извлечения возвращают ссылку на соответствующие объекты, являющиеся, соответственно, потоками вывода или ввода. Первые аргументы тоже должны быть ссылкой на поток. Второй параметр – ссылка на объект, выводящий или получающий информацию.

Эти функции не могут быть методами класса, для работы с которым они создаются. Если бы это было так, то левым аргументом, при вызове операции << (или >>), должен стоять объект, генерирующий вызов этой функции.

```
cout << "Моя строка"; a cout.operator << ("Моя строка"); // ошибка
```

Получается, что функция внесения, определяемая для объектов вашего класса, должна быть методом объекта cout.

Однако при этом можно столкнуться с серьезным ограничением, в случае, когда необходимо вывести (или ввести) значения защищенных членов класса. По этой причине, как правило, функции внесения и извлечения объявляют дружественными к создаваемому вами классу.

```
class _3d {
```

```
 double x, y, z;
public:
 _3d ();
 _3d (double initX, double initY, double initZ);
 double mod () {return sqrt (x*x + y*y +z*z);}
 double projection (_3d r) {return (x*r.x + y*r.y + z*r.z) / mod();}
 _3d& operator + (_3d& b);
 _3d& operator = (_3d& b);
 friend ostream& operator << (ostream& stream, _3d& obj);
 friend istream& operator >> (istream& stream, _3d& obj);
};
ostream& operator << (ostream& stream, _3d& obj) {
 stream << "x=" << obj.x<< "y=" << obj.y << "z=" << obj.z;
 return stream;
}
istream& operator << (istream& stream, _3d& obj) {
 stream >> obj.x >> obj.y >> obj.z;
 return stream;
}

main() {
 _3d vecA;
 cin >> vecA;
 cout << "My vector: " << vecA << "\n";
}
```

Хотя в функции внесения (или извлечения) может выполняться любая операция, лучше ограничить ее работу только выводом (или вводом) информации.

9.4 Функции библиотеки iostream

Основное отличие профессионального программиста от любителя заключается в том, что он проверяет все входные данные. В библиотеке iostream (таблица 9.2) для этого используются специальные методы класса ios и двух его наследников istream и ostream. Всего в этих классах находится порядка 25 методов, позволяющих получить информацию о состоянии объектов и управлять их поведением.

Таблица 9.2 – Функции библиотеки iostream

Имя функции	Действие
int good ()	возвращает 1, если ошибок не обнаружено
int eof ()	возвращает 1, если поток находится в состоянии "коней файла"
int fail ()	возвращает 1, если обнаружена восстановимая ошибка ввода-вывода (обычно, ошибка преобразования данных)
int bad ()	возвращает 1, если обнаружена невосстановимая ошибка ввода-вывода
int clear ()	сбрасывает состояние ошибки ввода-выводаф
int precision (int i)	устанавливает точность вывода чисел с плавающей точкой
int width (int i)	устанавливает ширину поля вывода

Таблица 9.3. Флаги

Флаг	Описание
left	выравнивание по левому краю
right	выравнивание по правому краю
dec, oct, hex	устанавливают базу для ввода-вывода
showbase	выводить показатель базы
showpoint	выводить десятичную точку для чисел с плав точкой
uppercase	16-ричные большие буквы
showpos	показать "+" для положительных целых чисел
scientific	установить экспон форму для чисел с плав точкой
fixed	fixed установить формат чисел с фиксированной плав точкой

Кроме того, вы можете создать свой собственные манипуляторы. Этому могут быть, по крайней мере, две причины. Во-первых, это может быть манипулятор, заключающий в себе некоторую часто

встречающуюся в программе последовательность манипуляторов. Во-вторых, для управления нестандартными устройствами ввода-вывода.

Создание непараметризованных манипуляторов не очень сложное занятие. Все, что для этого нужно, это определить функцию, имеющую прототип, похожий на следующий:

```
ios& manipul (ios&);
```

Если создается манипулятор только для вывода, ios заменяется на ostream, если для ввода – на istream.

В теле функции можно, что-то сделать с потоком, переданным в качестве параметров, а затем передать его в вызывающую программу на переданный функции аргумент типа поток.

Вот пример манипулятора для вывода с выравниванием по правому краю:

```
ostream & right(ostream &) {
 s << resetiosflags(ios::left) << setiosflags(ios::right);
 return s;
}
```

Функция манипулятор изменяет биты флагов потока, переданного ей в качестве аргумента. После того, как right установит биты флагов потока, последний будет выводиться в таком режиме до тех пор, пока флаги потока не будут изменены.

Таким образом, выражение:

```
cout << setw(20) << right<< 1234 << endl;
```

означает следующее. Манипулятор setw(20) установит поле вывода шириной 20 символов, манипулятор right установит выравнивание по правой границе поля вывода, выведется число, а манипулятор endl вызовет переход к следующей строке.

Параметризованные манипуляторы более сложны в написании, так как в этом случае необходимо создать класс, содержащий указатель на функцию – манипулятор и указатель на аргументы функции манипулятора. Макроопределения, которые облегчат процесс разработки параметризованных манипуляторов, содержатся в заголовочном файле iomanip.h.

9.5 Файловые и строковые потоки

Одной из самых привлекательных возможностей библиотеки iostream является то, что она одинаково совершенно работает с различными источниками данных для ввода и вывода: клавиатурой и эк-

раном, файлами и строками. Главное, что вы должны сделать, чтобы использовать строку или файл с библиотекой iostream,- это создать объект определенного типа.

Первостепенная задача при создании таких объектов - это организация буфера и связывание его с потоком.

При создании выходных файловых потоков вначале создается объект для буферизации типа filebuf, а затем этот объект связывается с новым объектом типа ostream для последующего вывода в этот новый объект.

```
filebuf mybuff; // создаем буферный объект
// увязывем этот буфер с файлом output для вывода
mybuff.open("output", ios::out);
ostream mycout(&mybuff); // новый потоковый объект
mycout <<12<<ends;
mybuff.close();
```

Аналогичные операции можно проделать при реализации извлечения из файла-буфера. Но можно поступить и по-другому, используя буферизированные файловые потоки трех видов:

ifstream – для ввода,
ofstream – для вывода,
fstream – для ввода и вывода.

Например,

```
int i;
// сразу создается буферизованный потоковый объект, связанный
// с файлом input
ifstream mycout("input");
mycin >> i;
```

Для выходных строковых потоков вы можно выделить буфер в программе или предоставить потоку возможность самому создать буфер динамически, но в таком случае придется заботиться о доступе к буферу, удалении и высвобождении выделенной под буфер памяти.

```
char mybuff[128];
ostrstream mycout (mybuff, sizeof(mybuff));
mycout << 123 << ends;
```

При этом mybuff примет значение "123".

Аналогично для ввода,

```
int i;
char mybuff[128]= "123";
istrstream mycin (mybuff, sizeof(mybuff));
mycin >> i;
```

После этого i примет значение 123. Для входных строковых потоков можно использовать символьный массив с завершающим нулем, либо указав точный размер.

Литература

1. Щуплов, В. В. Правила оформления дипломного проекта : методические указания к дипломному проекту для студентов специальности 1-36 04 02 «Промышленная электроника» днев. и заоч. форм обучения / В. В. Щуплов, Ю. В. Крышнев. – Гомель : ГГТУ им. П. О. Сухого, 2009.

2. Боровский, А. Н. Qt 4.7 Практическое программирование на C++ / А. Н. Боровский. – БВХ-Петербург. – 2012.

3. Кьоу, Д. Объектно ориентированное программирование / Д. Кьоу. – Питер. – 2005.

4. Лаптев, В. В. C++ объектно-ориентированное программирование / В. В. Лаптев. – Питер, 2008

5. Лафоре, Р. Объектно-ориентированное программирование в C++ / Р. Лафоре. – Санкт-Петербург : Питер, 2014

6. Богуславский, А. С ++ и компьютерная графика. Лекции и практикум по программированию на С ++ / А. С. Богуславский. – М. : Компьютер Пресс, 2003.

7. Волкова, И. А. Основы объектно-ориентированного программирования. Язык программирования C++ / А. В. Иванов, Л. Е. Карпов. – М. : МГУ, 2011.

8. Мейерс, С. Эффективное использование STL / С. Мейерс. – Gbnth 2001

9. Рудаков, А. В. Технология разработки программных продуктов. Практикум / А. В. Рудаков, Г. Н. Федорова. – М. : Академия, 2010.

Содержание

Сахарук Андрей Владимирович

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В СИСТЕМАХ УПРАВЛЕНИЯ

Учебно-методическое пособие по выполнению курсовой работы для студентов специальности 1-53 01 07 «Информационные технологии и управление в технических системах» дневной формы обучения

Подписано к размещению в электронную библиотеку ГГТУ им. П. О. Сухого в качестве электронного учебно-методического документа 17.04.25.

Рег. № 140Е.

http://www.gstu.by