НЕКОММЕРЧЕСКОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
УЧЕБНО-НАУЧНО-ПРОИЗВОДСТВЕННЫЙ КОМПЛЕКС
«МЕЖДУНАРОДНЫЙ УНИВЕРСИТЕТ КЫРГЫЗСТАНА»

«СОГЛАСОВАНО»
Проректор по учебно-административной работе НОУ УНПК «МУК»,
к.ю.н., Карабалаева С.Б.
_____
«_18_» ___11_____ 20__ г.

«УТВЕРЖДЕНО»
Ректор НОУ УНПК «МУК»,
к.т.н., доцент Савченко Е.Ю.
_____
«_19_» ___11_____ 2019 г.

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС

Название дисциплины: Математическая логика и теория алгоритмов

Направление: 710200 «Информационные системы и технологии»

Профиль: Информационные системы и технологии

Квалификация выпускника: Бакалавр

Форма обучения: Очная

Составитель (и): к.т.н., доцент Мусакулова Ж.А. _____

График проведения модулей
1-семестр

неделя	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
лекц. зан.	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
лаб. зан.	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	

«РАССМОТРЕНО»
На заседании кафедры
«КИСиУ»
НОУ УНПК «МУК»
Протокол № 2
от «14» 10 2019 г.
Зав. кафедрой
д.т.н., проф. Миркин Е.Л.
_____

«ОДОБРЕНО»
На заседании Учебно-методического объединения НОУ УНПК «МУК»
Протокол № 2
от «15» 11 2019 г.
Председатель Учебно-методического объединения
Матвеева Т.В. _____

Директор Научной библиотеки
НОУ УНПК «МУК»
Асанова Ж.Ш._____

Бишкек 2019г.

ОГЛАВЛЕНИЕ

АННОТАЦИЯ 3
УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ДИСЦИПЛИНЫ (МОДУЛЕЙ) 4
1. Пояснительная записка 4
1.1. Миссия и стратегия 4
1.2. Цель и задачи дисциплины (модулей) 4
1.3. Формируемые компетенции, а также перечень планируемых результатов обучения по дисциплине (модулю) 4
1.4. Место дисциплины (модулей) в структуре ООП ВПО 5
2. Структура и содержание дисциплины (модулей) 5
3. Конспект лекций 7
4. Информационные и образовательные технологии 7
5. Фонд оценочных средств для текущего, рубежного и итогового контролей по итогам освоению дисциплины (модулей) 9
5.1. Перечень компетенций с указанием этапов их формирования в процессе освоения дисциплины 9
5.2. Методические материалы, определяющие процедуры оценивания знаний, умений, навыков и (или) опыта деятельности 9
5.3. Описание показателей и критериев оценивания компетенций на различных этапах их формирования, описание шкал оценивания 10
6. Учебно-методическое и информационное обеспечение дисциплины 11
6.1. Список источников и литературы 11
6.2. Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимый для освоения дисциплины (модулей) 12
7. Материально-техническое обеспечение дисциплины 12
8. Приложения 12

АННОТАЦИЯ

Цель дисциплины (модулей) заключается в подготовке выпускника, имеющего специальные знания в области информационных технологий, для работы в области разработки программного обеспечения. Задачами дисциплины (модулей) являются: предоставление студентам навыков программирования на
языке высокого уровня; знакомство с основными принципами процедурного и объектно-ориентированного программирования; изучение данной дисциплины позволит студентам адаптироваться в среде программирования Visual C++, и освоить основные приемы и принципы программирования.

На изучение дисциплины отводится 150 часов. Рубежный контроль успеваемости проводится на 4, 8, 12, 16 неделях. Формы текущего контроля: опрос, проверка заданий, посещаемость. Форма рубежного контроля — модульная работа. Форма итогового контроля — экзамен.

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ДИСЦИПЛИНЫ (МОДУЛЕЙ)

1. Пояснительная записка

1.1. Миссия и стратегия

Миссия НОУ УНПК "МУК" – подготовка международно - признанных, свободно мыслящих специалистов, открытых для перемен и способных трансформировать знания в ценности на благо развития общества.
Видение НОУ УНПК «МУК»- создание динамичного и креативного университета с инновационными научно-образовательными программами и с современной инфраструктурой, способствующие достижению академических и профессиональных целей.
Стратегии развития - модернизация образовательной деятельности университета – совершенствование образовательного процесса в соответствии с требованиями Болонского процесса.

1.2. Цель и задачи дисциплины (модулей)

Цель дисциплины: предоставление студентам навыков программирования на
языке высокого уровня. Знакомство с основными принципами процедурного и объектно-ориентированного программирования. Изучение данной дисциплины позволит студентам адаптироваться в среде программирования Visual C++, и освоить основные приемы и принципы программирования.

Задачи дисциплины:

- получение знаний о современных методах в области программирования на языке C++;
- изучение синтаксиса языка программирования C++;
- получение базовых навыков и приёмов программирования;
- решение типовых задач обработки данных;

1.3. Формируемые компетенции, а также перечень планируемых результатов обучения по дисциплине (модулю)

Дисциплина (модуль) направлена на формирование следующих компетенций:

- общенаучными (ОК):
  - способен использовать базовые положения математических / естественных / гуманитарных / экономических наук при решении профессиональных задач (ОК-2);
- инструментальными (ИК):
  - владеть основными методами, способами и средствами получения, хранения и переработки информации, навыками работы с компьютером, как средством управления информацией, в том числе в глобальных компьютерных сетях и корпоративных информационных системах (ИК-5);
- профессиональными (ПК):
  - способен освоить методики использования программных средств для решения практических задач (ПК-2).

В результате освоения дисциплины обучающийся должен демонстрировать следующие результаты образования:

1. Знать:
   - законы логики высказываний и логики предикатов;
   - аксиоматизацию логики предикатов и основные теоремы: о дедукции, о

непротиворечивости, о полноте.

2. Уметь:
   - переводить выражения, сформулированные на обычном математическом языке, на язык логики предикатов;
   - строить выводы и доказательства формул;
   - применять изученные методы для доказательства логичности рассуждений;
   - писать программы машины для решения несложных задач.
3. Владеть:
   - основами математической логики и теории алгоритмов.

1.4. Место дисциплины (модулей) в структуре ООП ВПО

Дисциплина (модуль) является частью общенаучного цикла (блока) дисциплин учебного плана по направлению подготовки 710200 «Информационные системы и технологии».

2. Структура и содержание дисциплины (модулей)

Структура дисциплины (модулей) для очной формы обучения
Общая трудоемкость дисциплины составляет 5 кредитов, 150 ч., в том числе аудиторная работа обучающихся с преподавателем 85 ч., самостоятельная работа обучающихся 65 ч.

⁻ ʧ /	ʈ Θ ʟ н ʝ ʤ, ɼ ɖ ɦ ɥ ɖ ʧ ʤ ɖ ʤ ʳ	ʉ ʝ ʤ ʝ	ʅ ʝ н ʝ ʤ ɦ ʝ ʤ ʝ ɦ	ɹ ɖ н ʳ ʞ ɥ ʝ в ʤ ʦ ə ʢ ʤ ʁ ɥ Θ ʷ ɦ Θ ʤ ʦ ɦ ʟ ʦ ʷ ʟ ʝ ʤ ʹ ʦ Θ в ʦ ʟ ʟ ʞ ɦ ʟ ʟ ʞ ʟ ʦ ʟ н ʦ ʝ ʤ ʢ ʦ ɦ ʟ ʹ				ʌ ʦ ʦ ʤ ʳ ʟ ʝ ʢ ʞ ʰ ʝ ɢ ʦ ʢ ʦ ʤ ʟ ʦ ʦ ʤ ʷ ʞ ɦ ʧ ʝ ə Θ ʝ ʤ (ʧ ʦ ʤ ʝ ɦ ʝ ʤ ʝ ɦ ʟ ʦ Θ ʌ ʦ ʦ ʤ Θ ʧ ʦ ʦ ʤ ʝ ɤ ʞ ʟ Θ ʟ ʟ ʝ ɦ ʟ Θ ɥ ɦ ʝ ʤ ʝ ɦ ʟ ʦ Θ
				ʃ ʝ ʢ ɥ	ʉ ʝ / ʤ	ʉ ʈ ʉ	ʉ ʈ ʉ	
	Раздел 1.							
1	Введение в С++. Программа на языке С++.	1	1	2	3	3	1	опрос, проверка задания, посещаемость
2	Переменные. Константы. Концепция памяти. Арифметические и логические операции.	1	2	2	3	3	1	опрос, проверка задания, посещаемость
3	Библиотечные функции. Операторы ввода и вывода.	1	3	2	3	3	1	опрос, проверка задания, посещаемость
4	Управляющие структуры языка С++. Операторы условия.	1	4	2	3	3	1	Модуль 1
	Раздел 2.							

5	Компиляция и исполнение программы. Элементы языка С++.	1	5	2	3	3	1	опрос, проверка задания, посещаемость
6	Базовые типы данных. Ключевые слова.	1	6	2	3	3	1	опрос, проверка задания, посещаемость
7	Запуск. Компиляция.	1	7	2	3	3	1	опрос, проверка задания, посещаемость
8	Операторы присваивания. Инициализация переменных.	1	8	2	3	3	1	Модуль 2
	Раздел 3.							
9	Создание программ с использованием оператора множественного выбора switch.	1	9	2	3	3	1	опрос, проверка задания, посещаемость
10	Создание программ циклической структуры. Использование оператора for. Создание программ циклической структуры. Использование оператора while.	1	10	2	3	3	1	опрос, проверка задания, посещаемость
11	Создание программ с использованием одномерных массивов. Создание программ с использованием многомерных массивов.	1	11	2	3	3	1	опрос, проверка задания, посещаемость
12	Создание программ с многомерными символьными массивами.	1	12	2	3	3	1	Модуль 3
	Раздел 4.							
13	Функции, работающие со строками. Форматированный ввод – вывод printf(), scanf() Строки. Строковые переменные.	1	13	2	3	3	1	опрос, проверка задания, посещаемость

14	Инструменты потоков ввода/вывода. Ввод и вывод символов. Чтение одной строки. Чтение одиночного файла.	1	14	2	3	3	1	опрос, проверка задания, посещаемость
15	Указатели и строки. Переменные указатели.	1	15	2	3	3	1	опрос, проверка задания, посещаемость
16	Основы управления памятью компьютера. Статистические, динамические и автоматические переменные.	1	16	2	3	3	1	Модуль 4
17	Консультация	1	17	2	3	0	1	Консультация

3. Конспект лекций

Конспект лекций можно просмотреть в приложении 1.

4. Информационные и образовательные технологии

№ п/п	Наименование раздела	Виды учебной работы	Формируемые компетенции (указывается код компетенции)	Информационные и образовательные технологии
1	Раздел №1.	Лекция	ОК-2, ИК-5, ПК-2	Лекция-визуализация с применением слайд-проектора, Дискуссия, Лекция с разбором конкретных ситуаций
		Лабораторная работа	ОК-2, ИК-5, ПК-2	Дискуссия, Консультирование с разбором абстрактных ситуаций
		Самостоятельная работа	ОК-2, ИК-5, ПК-2	Использование электронного курса лекций, Консультирование и проверка заданий

				посредством электронной почты
2	Раздел №2.	Лекция	ОК-2, ИК-5, ПК-2	Лекция-визуализация с применением слайд-проектора, Дискуссия, Лекция с разбором конкретных ситуаций
		Лабораторная работа	ОК-2, ИК-5, ПК-2	Дискуссия, Консультирование с разбором абстрактных ситуаций
		Самостоятельная работа	ОК-2, ИК-5, ПК-2	Использование электронного курса лекций, Консультирование и проверка заданий посредством электронной почты
3	Раздел №3.	Лекция	ОК-2, ИК-5, ПК-2	Лекция-визуализация с применением слайд-проектора, Дискуссия, Лекция с разбором конкретных ситуаций
		Лабораторная работа	ОК-2, ИК-5, ПК-2	Дискуссия, Консультирование с разбором абстрактных ситуаций
		Самостоятельная работа	ОК-2, ИК-5, ПК-2	Использование электронного курса лекций, Консультирование и проверка заданий посредством электронной почты
4	Раздел №4.	Лекция	ОК-2, ИК-5, ПК-2	Лекция-визуализация с применением слайд-проектора, Дискуссия, Лекция с разбором конкретных ситуаций

		Лабораторная работа	ОК-2, ИК-5, ПК-2	Дискуссия, Консультирование с разбором абстрактных ситуаций
		Самостоятельная работа	ОК-2, ИК-5, ПК-2	Использование электронного курса лекций, Консультирование и проверка заданий посредством электронной почты

5. Фонд оценочных средств для текущего, рубежного и итогового контролей по итогам освоению дисциплины (модулей)

5.1. Перечень компетенций с указанием этапов их формирования в процессе освоения дисциплины

№ п/п	**Контролируемые разделы дисциплины (модулей)**	**Код контролируемой компетенции (компетенций)**	**Наименование оценочного средства**
1	Разделы №1, №2, №3, №4	ОК-2	опрос, выполнение лабораторных работ
2	Разделы №1, №2, №3, №4	ИК-5	опрос, выполнение лабораторных работ
3	Разделы №1, №2, №3, №4	ПК-2	опрос, выполнение лабораторных работ

5.2. Методические материалы, определяющие процедуры оценивания знаний, умений, навыков и (или) опыта деятельности

Форма контроля	Срок отчетности	Макс. количество баллов	
		За одну работу	Всего
Текущий контроль: - опрос	1, 2, 3, 4, 5 недели	8 баллов	До 40 баллов
- выполнение лабораторных работ	1, 2, 3, 4, 5 недели	6 баллов	До 30 баллов
- посещаемость	1, 2, 3, 4, 5 недели	0,2	10 баллов
Рубежный контроль: (сдача модуля)	5 неделя	100%×0,2=20 баллов	

Итого за I модуль			До 100 баллов

Форма контроля	Срок отчетности	Макс. количество баллов	
		За одну работу	Всего
Текущий контроль: - опрос	6, 7, 8, 9, 10 недели	8 баллов	До 40 баллов
- выполнение лабораторных работ	6, 7, 8, 9, 10 недели	6 баллов	До 30 баллов
- посещаемость	6, 7, 8, 9, 10 недели	0,2	10 баллов
Рубежный контроль: (сдача модуля)	10 неделя	100%×0,2=20 баллов	
Итого за II модуль			До 100 баллов

Форма контроля	Срок отчетности	Макс. количество баллов	
		За одну работу	Всего
Текущий контроль: - опрос	11, 12, 13, 14, 15 недели	8 баллов	До 40 баллов
- выполнение лабораторных работ	11, 12, 13, 14, 15 недели	6 баллов	До 30 баллов
- посещаемость	11, 12, 13, 14, 15 недели	0,2	10 баллов
Рубежный контроль: (сдача модуля)	15 неделя	100%×0,2=20 баллов	
Итого за III модуль			До 100 баллов
Итоговый контроль (экзамен)	Сессия	ИК = Бср × 0,8 + Бэкз × 0,2	

Полученный совокупный результат (максимум 100 баллов) конвертируется в традиционную шкалу:

Рейтинговая оценка (баллов)	**Оценка экзамена**
от 0 до 54	неудовлетворительно
от 55 до 69	удовлетворительно
от 70 до 84	хорошо
от 85 до 100	отлично

5.3. Описание показателей и критериев оценивания компетенций на различных этапах их формирования, описание шкал оценивания

Текущий контроль (0-80 баллов)

При оценивании посещаемости, опроса и выполнении лабораторных работ учитываются:

- посещаемость (10 баллов)
- степень раскрытия содержания материала (25 баллов);
- изложение материала (грамотность речи, точность использования терминологии и символики, логическая последовательность изложения материала (20 баллов);
- знание теории изученных вопросов, сформированность и устойчивость используемых при ответе умений и навыков (25 баллов).

Рубежный контроль (0 – 20 баллов)

При оценивании контрольной работы учитывается:

- полнота выполненной работы (задание выполнено не полностью и/или допущены две и более ошибки или три и более неточности) – 10 баллов;
- обоснованность содержания и выводов работы (задание выполнено полностью, но обоснование содержания и выводов недостаточны, но рассуждения верны) – 5 баллов;
- работа выполнена полностью, в рассуждениях и обосновании нет пробелов или ошибок, возможна одна неточность – 5 баллов.

Итоговый контроль (экзаменационная сессия) – ИК = Бср × 0,8 + Бэкз × 0,2

При проведении итогового контроля обучающийся должен ответить на 3 вопроса (два вопроса теоретического характера и один вопрос практического характера).

При оценивании ответа на вопрос теоретического характера учитывается:

- теоретическое содержание не освоено, знание материала носит фрагментарный характер, наличие грубых ошибок в ответе (0 баллов);
- теоретическое содержание освоено частично, допущено не более двух-трех недочетов (10 баллов);
- теоретическое содержание освоено почти полностью, допущено не более одного-двух недочетов, но обучающийся смог бы их исправить самостоятельно (20 баллов);
- теоретическое содержание освоено полностью, ответ построен по собственному плану (30 баллов).

При оценивании ответа на вопрос практического характера учитывается:

- ответ содержит менее 20% правильного решения (0-9 баллов);
- ответ содержит 21-89 % правильного решения (10-39 баллов);
- ответ содержит 90% и более правильного решения (40 баллов).

6. Учебно-методическое и информационное обеспечение дисциплины

6.1. Список источников и литературы

Литература:

- Основная:
  1. Б. Страуструп. Язык программирования С++. Издательство Бином. Москва. (любое издание).
  2. В.В. Подбельский. Язык С++. Издательство Финансы и статистика. г. Москва. 1999

г.
3. Д. Нейл, Ч.Уимз, М. Хедингтон. Учебник. Программирование на C++. Москва. 2000 г.
4. Ю. Тихомиров. Visual C++6. Новые возможности для программистов. Издательство BHV. г. Санкт – Петербург. 1999 г.

¶ Дополнительная:

1. Б.И. Березин, С.И. Березин. Начальный курс С и С++. Издательство Диалог МИФИ. Москва (любое издание).
2. Программирование на С++. Учебное пособие. Под редакцией А.Д. Хомоненко. Издательство КОРОНА Принт. Санкт – Петербург. 1999 г.
3. Э.А. Ишкова. С++ начала программирования. Издательство Бином. Москва. 2007 г.

6.2. Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимый для освоения дисциплины (модулей)

¶ realcoding.net
¶ ishodniki.ru
¶ citforum.ru
¶ http://www.iprbookshop.ru/
¶ http://kyrlibnet.kg/ru/
¶ http://biblioteka.kg/

7. Материально-техническое обеспечение дисциплины

Для изучения дисциплины, необходимо следующее оборудование: ЭВМ, проектор.

Требования к аудитории: компьютерный класс, имеющий ЭВМ в количестве идентичном количеству обучающихся, ЭВМ для преподавателя с подключенным проектором, наличие доски и средств для отображения/удаления информации на доске (мел/ветошь, маркер/губка).

8. Приложения

Приложение 1

Лекция 1. Введение в язык С++. Типы данных. Математические операции. Сокращенная запись операций. Инкремент/декремент. (см. слайды П_1_1_Синтаксис)

С++ — компилируемый статически типизированный язык программирования общего назначения. Являясь одним из самых популярных языков программирования, С++ широко используется для разработки программного обеспечения. Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также развлекательных приложений (например, видеоигры). Существует несколько реализаций языка С++ — как бесплатных, так и коммерческих. При создании С++ Бьёрн Страуструп стремился сохранить совместимость с языком С. Множество программ, которые могут одинаково успешно транслироваться как компиляторами С, так и компиляторами С++, довольно велико — отчасти благодаря тому, что синтаксис С++ был основан на синтаксисе С. Переменные и типы данных Переменные — это участки памяти, используемые программой для хранения данных. Прежде чем использовать переменную, ее необходимо предварительно объявить глобально (вне функций) или локально (внутри функции). В большинстве случаев

объявление переменной является сразу и ее определением. Глобальные переменные видны внутри всех функций в файле, а локальные переменные видны только внутри той функции, в которой они объявлены. Для объявления переменной используется следующий формат: [<Спецификатор>][<Модификатор>] <Тип> <Переменная1>[=<Значение1>] [, ..., <ПеременнаяN>[=<ЗначениеN>]]; Именование переменных Каждая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр и знака подчеркивания, причем имя переменной не может начинаться с цифры. При указании имени переменной важно учитывать регистр букв: x и X — разные переменные. В качестве имени переменной нельзя использовать ключевые слова. Следует учитывать, что в некоторых компиляторах могут быть определены дополнительные ключевые слова.
Запоминать все ключевые слова нет необходимости, так как в редакторе эти слова подсвечиваются. Любая попытка использования ключевого слова вместо названия переменной приведет к ошибке при компиляции. Помимо ключевых слов следует избегать совпадений со встроенными идентификаторами.
Правильные имена переменных: x, y1, str_name, strName. Неправильные имена переменных: 1y, ИмяПеременной. Последнее имя неправильное, так как в нем используются русские буквы. Хотя на самом деле такой вариант также будет работать, но лучше русские буквы все же не применять. Типы данных В языке C++ определены следующие элементарные типы данных: void — означает отсутствие типа. Используется в основном чтобы указать, что функция не возвращает никакого значения, а также для передачи в функцию данных произвольного типа. Пример объявления функции, которая не возвращает значения и принимает указатель на данные произвольного типа: void free(void *Memory); bool — логический тип данных. Может содержать значения true (соответствует числу 1) или false (соответствует числу 0). Пример объявления переменной: bool is_int; char — код символа. Диапазон значений от -128 до 127. Пример объявления переменной: char ch; wchar_t — двухбайтовый символ. Пример объявления переменной: wchar_t wch; int — целое число со знаком. Диапазон значений зависит от разрядности операционной системы. В 16-битной операционной системе диапазон от -32 768 до 32 767. В 32-битной операционной системе диапазон от -2 147 483 648 до 2 147 483 647. Пример объявления переменной: int x; float — вещественное число. Пример объявления переменной: float y; double — вещественное число двойной точности. Пример объявления переменной: double z; Перед элементарным типом данных могут быть указаны следующие модификаторы или их комбинация: signed — указывает, что символьный или целочисленный типы могут содержать отрицательные значения. Тип signed char соответствует типу char и может содержать значения от -128 до 127. Пример объявления переменной: signed char ch; Тип signed int соответствует типу int. Пример объявления переменной: signed int x; unsigned — указывает, что символьный или целочисленный типы не могут содержать отрицательные значения. Тип unsigned char может содержать значения от 0 до255. Пример объявления переменной: unsigned char ch; Диапазон значений для типа unsigned int зависит от разрядности операционной системы. В 16-битной операционной системе диапазон от 0 до 65 535. В 32-битной операционной системе диапазон от 0 до 4 294 967 295. Пример объявления переменной: unsigned int x; short — может быть указан перед целочисленным типом. Диапазон значений у типов short int и signed short int от -32 768 до 32 767. Пример объявления переменных: short int x; signed short int y; Диапазон значений у типа unsigned short int от 0 до 65 535. Пример объявления переменной: unsigned short int x;
long — может быть указан перед целочисленным типом и типом double. Диапазон значений у типов long int и signed long int от -2 147 483 648 до 2 147 483 647. Пример объявления переменных: long int x; signed long int y; Диапазон значений у типа unsigned long int от 0 до 4 294 967 295. Пример объявления переменной: unsigned long int x; Ключевое слово long перед целым типом может быть указано дважды. Диапазон значений у типов long long int и signed long long int от9 223 372 036 854 775 808 до 9 223 372 036 854 775 807. Пример объявления

переменных: long long int x; signed long long int y; Диапазон значений у типа unsigned long long int может быть от 0 до 18 446 744 073 709 551 615. Пример объявления переменной: unsigned long long int x; Ключевое слово long можно указать также перед типом double. Пример объявления переменной: long double z; При использовании модификаторов тип int подразумевается по умолчанию, поэтому тип int можно не указывать. Пример объявления переменных: short x; // Эквивалентно: short int x; long y; // Эквивалентно: long int y; signed z; // Эквивалентно: signed int z; unsigned k; // Эквивалентно: unsigned int k; В одной инструкции можно объявить сразу несколько переменных, указав из через запятую после названия типа данных: int x, y, z; Если переменная может изменять свое значение извне, то перед модификатором указывается ключевое слово volatile. Это ключевое слово предотвращает проведение оптимизации программы, при котором предполагается, что значение переменной может быть изменено только в программе. После объявления переменной под нее выделяется определенная память, размер которой зависит от используемого типа данных и разрядности операционной системы. Например, тип int в 16-битной системе занимает 16 бит, в 32-битной — 32 бита, а в 64битной — 64 бита. Чтобы сделать код машинонезависимым следует определять размер типа с помощью оператора sizeof. Оператор имеет два формата: <Размер> = sizeof <Переменная>; <Размер> = sizeof (<Тип данных>); Пример: int x; std::cout << sizeof x << std::endl; std::cout << sizeof (x) << std::endl; std::cout << sizeof (bool) << std::endl; Математические операции

Знак / всегда означает деление. Однако если с обеих сторон от этого знака стоят целые величины (константы, переменные или их комбинации), он означает целочисленное деление. Если в результате такого деления получается остаток, C++ его отбрасывает. Знак % означает выделение остатка при целочисленном делении. Эта операция требует, чтобы с обеих сторон от ее знака стояли целые величины. Присваивание В одной строке может стоять больше одной операции присваивания =. Пример. a=b=c=d=100; Знак = всегда означает: "переменной слева присвоить значение, стоящее справа ". Операция выполняется справа налево. Поэтому первой значение 100 получает переменная d, затем c, b и a. Знак присвоить может стоять даже внутри математического выражения: value=5+(r=9-c) Присваивание имеет более высокий приоритет, чем сложение и вычитание. Поэтому сначала переменной r будет присвоено значение 9-c. А затем переменная value получит значение 5+9-c. Составное присваивание При написании программы часто требуется изменить значение переменной. Например, требуется взять текущее значение переменной, прибавить или умножить это значение на какое-то выражение, а затем присвоить это значение той же переменной. Такие операции выполняют операторы составного присваивания.

Преобразование типов Чаще всего Вам не надо беспокоиться об автоматическом преобразовании типов. Однако, если Вы смешивает беззнаковые переменные с переменными других типов, могут возникнуть проблемы. Точность результата может быть потеряна. Поэтому необходимо произвести временное изменение типов. Формат: (тип данных) выражение Тип данных - это любой допустимый тип. Выражение - переменная, константа или выражение. Пример. v=(double)age*f; - переменная age временно преобразуется к типу с плавающей точкой двойной

точности и умножается на переменную f. Операции инкремента (++) и декремента (--) В языке C++ предусмотрены две уникальные операции, которые увеличивают или уменьшают значение переменной на 1.

Префиксный и постфиксные операции различаются приоритетом. Префиксные операции имеют самый большой приоритет и выполняются до любой другой операции. Постфиксные операции имею самый маленький приоритет и выполняются после всех остальных операции.

Лекция 2. Управляющие конструкции. Статические массивы. Функции. (см. слайды П_1_2_Управляющие конструкции)

Фигурные скобки Фигурные скобки позволяют объединить несколько элементарных

операторов в один составной оператор, или блок. Во всех синтаксических конструкциях составной оператор можно использовать вместо простого. В Си в начало блока можно помещать описания локальных переменных. Локальные переменные, описанные внутри блока, создаются при входе в блок и уничтожаются при выходе из него. В C++ локальные переменные можно описывать где угодно, а не только в начале блока. Тем не менее, они, так же как и в Си, автоматически уничтожаются при выходе из блока. Приведем фрагмент программы, обменивающий значения двух вещественных переменных: double x, y; . . . { double tmp = x; x = y; y = tmp; } Здесь, чтобы обменять значения двух переменных x и y, мы сначала запоминаем значение x во вспомогательной переменной tmp. Затем в x записывается значение y, а в y - сохраненное в tmp предыдущее значение x. Поскольку переменная tmp нужна только внутри этого фрагмента, мы заключили его в блок и описали переменную tmp внутри этого блока. По выходу из блока память, занятая переменной tmp, будет освобождена. Оператор if Оператор if ("если") позволяет организовать ветвление в программе. Он имеет две формы: оператор "если" и оператор "если...иначе". Оператор "если" имеет вид if (условие) действие; оператор "если...иначе" имеет вид if (условие) действие1; else действие2; Условная операция ? В C++ есть условная операция, очень похожая на условный оператор выбора if else. Условная операция "? :" называется тернарной операцией (то есть трёхместная (имеет три операнда), единственная в C++). Форма записи тернарной операции в C++ "условие" ? "выражение1" : "выражение 2"; Если условие истинно, то выполняется выражение 1, иначе (условие ложно) выполняется выражение 2. Пример: a > b ? cout << a : cout << b; // если a > b, то выполняется cout << a, иначе выполняется cout << b Статические массивы Массив является одним из ключевых понятий программирования. Итак, массив — это несколько значений одного типа, расположенных в памяти последовательно. Зная
местоположение первого из этих элементов, можно обратиться к любому из них. Порядковый номер элемента в массиве называется индексом. Массивы — это та часть языка Си, которая не подверглась изменениям при эволюционировании языка в C++. Поэтому их объявление и работа с ними на обоих этих языках совпадает. Чтобы создать в памяти новый массив, используется такая запись: int m[10]; int — это тип элементов массива, одинаковый для всех них. Конечно, Вы можете использовать любой другой тип (кроме void, ссылок и типов с неизвестным размером), чтобы задавать массивы. Квадратные скобки обозначают, что это массив, целое число в скобках (обязательно константа, известная на момент компиляции) — его размер, т.е. количество элементов. m — это имя нашей переменной-массива. Цикл while Конструкция цикла "пока": while (условие) действие; Цикл while называют циклом с предусловием, поскольку условие проверяется перед выполнением тела цикла. Цикл while выполняется следующим образом: сначала проверяется условие. Если оно истинно, то выполняется действие. Затем снова проверяется условие ; если оно истинно, то снова повторяется действие, и так до бесконечности. Цикл завершается, когда условие становится ложным. Если условие ложно с самого начала, то действие не выполняется ни разу. Это очень облегчает программирование и делает программу более надежной, поскольку исключительные ситуации автоматически правильно обрабатываются. При ошибке программирования цикл может никогда не кончиться. Чтобы избежать этого, следует составлять программу таким образом, чтобы некоторая ограниченная величина, от которой прямо или косвенно зависит условие в заголовке цикла, монотонно убывала или возрастала после каждого выполнения тела цикла. Это обеспечивает завершение цикла. Оператор do...while do...while - это цикл с постусловием. Цикл do ... while - цикл с неизвестным числом повторений. В отличие от цикла while условие проверяется в конце цикла. Поэтому тело цикла выполняется хотя бы один раз do {блок из одного или нескольких операторов} while (условие) Цикл for Популярный в других языках программирования арифметический цикл в языке Си реализуется с помощью цикла for. Он выглядит следующим образом: for (инициализация;

условие продолжения; итератор) тело цикла; Инициализация выполняется один раз перед первой проверкой условия продолжения и первым выполнением тела цикла. Условие продолжения проверяется перед каждым выполнением тела цикла. Если условие истинно, то выполняется тело цикла, иначе цикл завершается. Итератор выполняется после каждого выполнения тела цикла (перед следующей проверкой условия продолжения). Поскольку условие продолжения проверяется перед выполнением тела цикла, цикл for является, подобно циклу while, циклом с предусловием. Если условие продолжения не выполняется изначально, то тело цикла не выполняется ни разу, а это хорошо как с точки зрения надежности программы, так и с точки зрения простоты и эстетики (поскольку не нужно отдельно рассматривать исключительные случаи). Рассмотрим пример суммирования массива с использованием цикла for: double a[100]; // Массив a содержит не более 100 элементов
int n; // Реальная длина массива a (n <= 100) double sum; // Переменная для суммы эл-тов массива . . . sum = 0.0; for (int i = 0; i < n; i++) { sum += a[i]; // Увеличиваем сумму на a[i] } Оператор switch Оператор switch - оператор выбора. Формат: switch (выражение) { case (выражение 1): блок из одного или нескольких операторов 1; break; case (выражение2): блок из одного или нескольких операторов 2; break; case (выражение3): блок из одного или нескольких операторов 3; break; default: блок из одного или нескольких операторов n; break; } Выражением могут быть любое выражение, константа или переменная типа int или char. Оператор switch используется вместо нескольких операторов if else. Если выражение принимает значение выражение1, то выполняется блок из одного или нескольких операторов 1, если выражение принимает значение выражение2, то выполняется блок из одного или нескольких операторов 2, и т.д. Если выражение не совпадет ни с выражением под номером, то выполняется блок из одного или нескольких операторов n. Строка default - необязательная и может отсутствовать. Функции в C++ Функция - это группа операторов у которой есть имя. Основные концепции: • Функции группируют связанные операторы для выполнения определенной задачи. • Программа вызывает функцию, обращаясь к ее имени, за которым следуют круглые скобки, например beep (). • После завершения обработки большинство функций возвращают значение определенного типа, например int или float, которое программа может проверить или присвоить переменной. • Программы передают параметры (информацию) функциям, например имя, возраст или оклад служащего, заключая параметры в круглые скобки, которые следуют за именем функции. • C++ использует прототипы функций для определения типа возвращаемого функцией значения, а также количества и типов параметров, передаваемых функции. Функция C++ по структуре подобна функции main. Другими словами, имя функции предваряется ее типом, а за ним следует список параметров, описание которых появляется в скобках. Вы группируете операторы функций внутри левой и правой фигурных скобок, как
показано ниже: тип_возврата имя_функции (список_параметров) { тело функции; } После того как создано определение функции, её можно вызвать. int sum (int a, int b) { int c; c = a + b; return c; } int main() { int s; s = sum(2,2); // вызов функции cout << s; return 0; } Если на месте возвращаемого типа, в определении (и в объявлении) функции move стоит ключевое слово void (void - пусто, пустой), это значит, что функция не возвращает никакого значения. Следовательно не требуется и оператор return, так как функции нечего вернуть в вызывающее окружение.
Лекция 3. Понятие ООП. Класс. Инкапсуляция. Зоны видимости. Поля. (см. слайды П_1_3_ООП)
ООП (объектно-ориентированное программирование) — парадигма программирования, в которой основными понятиями являются объекты и классы. В известном смысле объект представляет собой сущность. Программа обычно использует переменные для хранения информации о различных реально существующих сущностях, например служащих, книгах и даже файлах. При объектно-ориентированном программировании вы фокусируетесь на

предметах, образующих систему, и операциях, которые вы должны выполнять над этими предметами. Классы Механизм классов в C++ позволяет пользователям определять собственные типы данных. По этой причине их часто называют пользовательскими типами. Класс может наделять дополнительной функциональностью уже существующий тип. Определение класса состоит из двух частей: заголовка, включающего ключевое слово class, за которым следует имя класса, и тела, заключенного в фигурные скобки. После такого определения должны стоять точка с запятой: class Screen { /* ... */ }; Внутри тела объявляются данные-члены и функции-члены и указываются уровни доступа к ним. Таким образом, тело класса определяет список его членов. Каждое определение вводит новый тип данных. Даже если два класса имеют одинаковые списки членов, они все равно считаются разными типами:
class First { int memi; double memd; };
class Second { int memi; double memd; };
class First obj1; Second obj2 = obj1; // ошибка: obj1 и obj2 имеют разные типы Тело класса определяет отдельную область видимости. Объявление членов внутри тела помещает их имена в область видимости класса. Наличие в двух разных классах членов с одинаковыми именами – не ошибка, эти имена относятся к разным объектам. (Подробнее об областях видимости классов мы поговорим в разделе 13.9.) После того как тип класса определен, на него можно ссылаться двумя способами: • написать ключевое слово class, а после него – имя класса. В предыдущем примере объект obj1 класса First объявлен именно таким образом; • указать только имя класса. Так объявлен объект obj2 класса Second из приведенного примера. Оба способа сослаться на тип класса эквивалентны. Первый заимствован из языка C и остается корректным методом задания типа класса; второй способ введен в C++ для упрощения объявлений. Данные-члены класса объявляются так же, как переменные. Функции-члены класса объявляются в его теле. Это объявление выглядит точно так же, как объявление функции в области видимости пространства имен. Функции-члены отличаются от обычных функций следующим: • функция-член объявлена в области видимости своего класса, следовательно, ее имя не
видно за пределами этой области. К функции-члену можно обратиться с помощью одного из операторов доступа к членам – точки (.) или стрелки (->): • функции-члены имеют право доступа как к открытым, так и к закрытым членам класса, тогда как обычным функциям доступны лишь открытые. Конечно, функциичлены одного класса, как правило, не имеют доступа к данным-членам другого класса. Функция-член может быть перегруженной. Однако она способна перегружать лишь другую функцию-член своего класса. По отношению к функциям, объявленным в других классах или пространствах имен, функция-член находится в отдельной области видимости и, следовательно, не может перегружать их. Зоны видимости/модификаторы доступа Почти каждому члену класса можно установить модификатор доступа (за исключением статических конструкторов и некоторых других вещей). В большинстве объектно-ориентированных языков программирования поддерживаются следующие модификаторы доступа: • private (закрытый, внутренний член класса) — обращения к члену допускаются только из методов того класса, в котором этот член определён. Любые наследники класса уже не смогут получить доступ к этому члену. • protected (защищённый, внутренний член иерархии классов) — обращения к члену допускаются из методов того класса, в котором этот член определён, а также из любых методов его классов-наследников. • public (открытый член класса) — обращения к члену допускаются из любого кода. Проблема поддержания правильного состояния переменных актуальна и для самого первого момента выставления начальных значений. Для этого в классах предусмотрены специальные методы/функции, называемые конструкторами. Ни один объект (экземпляр класса) не может быть создан иначе, как путем вызова на исполнение кода конструктора, который вернет вызывающей стороне созданный и правильно заполненный экземпляр класса. Во многих языках программирования тип данных «структура», как и класс, может содержать

переменные и методы, но экземпляры структур, оставаясь просто размеченным участком оперативной памяти, могут создаваться в обход конструкторов, что запрещено для экземпляров классов (за исключением специальных исключительных методов обхода всех подобных правил ООП, предусмотренных в некоторых языках и платформах). В этом проявляется отличие классов от других типов данных — вызов конструктора обязателен. Пример (листинг class1.cpp): #include <cstdlib> #include <iostream>
using namespace std;
class clStudent{ public: int spec; char name[100]; int marks[10]; clStudent(); clStudent(int specCode); void study(); void output(); ~clStudent(); }; clStudent::~clStudent(){} clStudent::clStudent(){
cout<< "New student name - "; cin>>name; }
clStudent::clStudent(int specCode){ cout<< "New student name - "; cin>>name; spec = specCode; }
void clStudent::study(){ for (int i=0; i<10; i++){ cout<<"input mark - "; cin>> marks[i]; } } void clStudent::output(){ cout<< "\n"; cout<< "Name is "<< name<<"\n"; cout<< "Marks are: "; for (int i=0; i<10; i++){ cout<<marks[i]<<" "; } } int main(int argc, char *argv[]) { clStudent specStudent(230100); specStudent.study(); specStudent.output(); cout<< "\n"; system("PAUSE"); return EXIT_SUCCESS; }
Лекция 4. Адреса и указатели. Динамические массивы. Friend-методы. (см. слайды П_1_4_Указатели_friend)
Переменная Что такое переменная? Переменная – это определенное место в памяти, имеющее имя, по которому мы его используем. Переменная имеет значение (текущее значение), грубо говоря, это то, что записано сейчас по определенному месту в памяти. Переменная также имеет тип, описывающий, какого рода значения в ней хранятся. Переменные устроены таким образом, что мы можем работать с ними по их имени, не задумываясь, где на самом деле в памяти и как они хранятся – это их главная идея. Прежде чем использовать переменную, ее надо объявить и определить ее тип. Одновременно с этим можно задать начальное значение, а можно установить значение переменной после, в любой момент. float f; // объявляем перем. f типа float char s = ‘w’; // объяв. перем. s типа char и присв. ей знач. ‘w’ f = -3.2; // перем. f присв. значение -3.2 int a = 5; // объявляем перем. a типа int и присв. ей значение 5 int b = 6; // объявляем перем. b типа int и присв. ей значение 6 int t; // объявляем перем. t типа int t = a; // переменной t присваиваем значение переменной a a = b; // переменной a присваиваем значение переменной b b = t; // переменной b присваиваем значение переменной t Во многих языках это было бы практически все, что можно делать с переменными, как с ними работать. Однако в C++, как в языке низкого уровня, с переменными можно работать и по-другому, а именно – косвенно. Для того, чтобы в этом разобраться, нужно для начала разобраться в том, что представляет из себя память, в которой переменные хранятся. Ведь переменные имеют различные типы, а память одна – как в одной и той же памяти хранятся и целые числа, и дробные числа, и символы, и так далее. Память Память – это нумерованная последовательность числовых двоичных байтов. Именно нумерованная (то есть каждый байт имеет номер), именно последовательность (то есть каждый следующий байт имеет номер, отличающийся на единицу), именно двоичных (то есть байт содержит 8 двоичных битов, содержащих, в свою очередь, 0 или 1), именно байтов (то есть можно обратиться только[1] к байту целиком, но не к отдельному биту), и именно числовых (то есть каждый байт представляет из себя число). Каждый байт этой последовательности имеет свой номер и текущее значение, которое, как несложно догадаться, в двоичном виде представляет из себя число от 00000000 до 11111111, в шестнадцатеричном виде – число от 00 до FF, и в десятичном виде – число от 0 до 255:
Как уже говорилось выше, каждая переменная занимает некоторое место в памяти. Разные типы требуют различного количества байтов. Еще раз: переменная – это определенное место

в памяти, имеющее имя, по которому мы его используем. Теперь мы можем сказать более точно. Определенное место – это некоторая определенная непрерывная последовательность байтов в памяти:
Что мы здесь видим? Во-первых, все переменные занимают разное количество (подряд идущих!) байтов. Во-вторых, некоторые байты могут быть не заняты, но, смотря только на значения этих байтов, не скажешь, заняты они или нет, и значениями каких типов заняты. Незанятые байты могут иметь произвольные значения, оставшиеся от каких-то переменных, которые там раньше размещались, но теперь не размещаются. Иногда мы такие байты будем помечать символами '?', но надо обязательно держать в уме, что там не вопросы находятся, а какие-то реальные значения. И, в-третьих, и в самых главных, все переменные хранятся в той же самой памяти, которая состоит из числовых байтов. Внимание, вопрос. Как различные типы данных могут храниться в одной и той же памяти? Как в числовой памяти можно хранить символ 'w'? Как в двух ячейках, каждая из которых может содержать число от 0 до 255, можно хранить число 258? Для каждого типа данных известно, сколько байтов в памяти занимает переменная его типа (для char – один байт, для int – два байта, и так далее), и, самое главное, – как любое значение этого типа представить в виде последовательности числовых байтов. Для каждого значения char, например, выделяется один байт, и в этом числовом байте хранится номер этого символа в специальной таблице ascii. Адрес переменной Адрес переменной – это, грубо говоря, номер первого байта, занимаемого этой переменной в памяти. В вышеприведенном примере, адрес переменной s есть 1, адрес переменной a есть 2, адрес переменной b есть 4. Зная адрес некоторой переменной, и зная ее тип, можно точно узнать, с какого по какой байты в памяти эта переменная занимает. А именно, байты с номерами с адрес(переменной) по адрес(переменной) + размер(переменной) - 1. Для каждой переменной можно узнать ее адрес, с помощью операции &. Адрес – это, грубо говоря, номер, то есть число. Но не все операции над адресом допустимы и, что важнее, осмысленны. Приведу такую аналогию. Адрес – это, например, номер почтового ящика. Если рассмотреть ящик номер 5, то его адрес – это, соответственно 5. К этому адресу я могу прибавить какое-то число, например, 2, и получить новый адрес, а именно 7. Если рассмотреть ящик номер 5 и ящик номер 8, то я могу из адреса второго вычесть адрес первого и получить уже не адрес, но число, равное количеству ящиков между ящиком номер 5 и ящиком номер 8. Но складывать два адреса нельзя, это бессмысленная операция. Итак, над адресами переменных возможны следующие операции: адрес2 – адрес1 => число1 – количество байтов между байтами по адресам адрес1 и адрес2; адрес1 + число1 => адрес2 – адрес байта, отстоящего от байта по адресу адрес1 на число1 байтов int a = 5; int b = 6; if (&b - &a > 10) /* переменные далеко друг от друга */ if (&a + sizeof(a) == &b) /* перем. b идет в памяти сразу за a */ Важный вопрос – а если мы захотим получить адрес некоторой переменной и сохранить его временно где-то в памяти, какого размера область памяти нам надо выделить? Или подобный вопрос: в переменную какого типа можно записать адрес? Если адрес – это
номер, то ответ зависит от того, как много может быть таких адресов? В общем-то, адресов может быть столько, сколько ячеек в памяти, а это может быть и 10^9, и даже больше. Переменные-указатели Что такое указатель? Указатель – это «всего-лишь» переменная, значением которой является адрес. То есть это именно такая переменная, в которой можно сохранить адрес какой-то другой переменной. Представьте себе переменную как почтовый ящик с бумажкой внутри, на которой что-то написано. Адрес этой переменной – это номер почтового ящика. Так вот, указатель – это почтовый ящик с бумажкой внутри, на которой написан номер какого-то другого почтового ящика. Зачем это в принципе может быть нужно? Если продолжать аналогию с почтовыми ящиками, то можно привести такой пример. Пусть у меня есть два почтовых ящика, ящик1 и ящик2. Я хочу, чтобы мои письма доставлялись в какие-то другие почтовые ящики. Первый вариант этого добиться – это поставить перед почтальоном задачу вида: «Доставь письмо из ящика1 в ящик5». Завтра мне понадобится

доставить письмо в ящик6, и мне придется менять задание почтальону, и в конце концов он все перепутает (грубо говоря – нехорошо каждый раз менять задание, это требует дополнительной информации почтальону). Другой вариант этого добиться – это поставить перед почтальоном постоянную задачу вида: «Доставь письмо из ящика1 в ящик, номер которого находится в ящике2». Да, это чуть более сложная задача для почтальона, но она постоянная, она не меняется, и он всегда может действовать по одному и тому же шаблону. Второй вариант называется косвенной адресацией, и ящик2, как вы уже догадались – это указатель, то есть такая переменная, которая содержит адрес другой переменной. Так же как переменные различаются типами, указатели тоже бывают разными. Так при объявлении указателя обязательно задается, на переменную какого типа это указатель. То есть, по сути – адрес переменной какого типа будет содержаться в этом указателе. Это может показаться лишним, потому что, в любом случае, указатель – это адрес первого байта в памяти, которую занимает переменная (вне зависимости от того, это переменная какого типа). Но так или иначе, этот тип нужно указывать. int a = 5; // объявляем перем. a типа int и присв. ей значение 5 int *ptr; // объявляем ptr как перем.-указатель на перем. типа int

В данном случае мы объявляем новую переменную ptr, в которой будет храниться адрес какой-то переменной типа int. Другими словами, в переменной ptr будет храниться адрес первого байта области в памяти, в которой хранится переменная типа int. Пока мы только объявили эту переменную, и она пока не содержит никакого осмысленного значения (не содержит осмысленного адреса). Занесем теперь в ptr адрес, например, переменной a. ptr = &a;

Так же, как с самой переменной a, мы могли значение указателя (значением указателя является адрес!) установить сразу при объявлении указателя:

int *ptr = &a; int *ptr2 = &a; int *ptr3 = ptr;

В приведенном примере создается три указателя на переменную типа int. В переменные-указатели ptr и ptr2 сразу заносится адрес переменной a, а в переменную ptr3 заносится значение указателя ptr, а раз значением указателя является адрес, и в ptr к тому моменту находится адрес переменной a – в переменной ptr3 тоже окажется адрес переменной a. Итак, пусть есть имеется переменная ptr, которая является указателем на переменную типа int. Как работают и для чего используют переменные-указатели? int a = 5; int *ptr = &a; // int *ptr; - объявление ptr как новой переменной-указателя // ptr – переменная-указатель, ее значением является адрес // *ptr – значение по адресу, хранимому в переменной ptr То есть самой главной операцией у указателей является *. При объявлении указателя, символ * показывает, что это именно не переменная типа int, а указатель на переменную типа int. А при использовании указателя, символ * перед именем указателя позволяет получить не сам адрес, а значение по этому адресу. Рассмотрим такой пример: int a = 5; int *ptr = &a; int b = a; // a => 5 // ptr => адрес a // b => 5 // *ptr => 5

Что теперь произойдет, если мы в переменную a запишем 6?

a = 6;

Как видим, значение ptr и значение b не изменились, но:

// a => 6 // ptr => адрес a // b => 5 // *ptr => 6 (*ptr) раньше было равно 5, а теперь стало равно 6. Операции с указателями Как уже говорилось выше, указатели допускают несколько видов операций над ними. В частности разность двух указателей есть число, а указатель плюс число есть опять указатель. Разберемся поподробнее. char c1 = ‘h’; char c2 = ‘e’; char c3 = ‘l’; char c4 = ‘l’; char c5 = ‘o’; char c6 = 0; // C++ позволяет вот так символ рассматривать как число от // 0 до 255 и оперировать с ним, как с числом char *ptr = &c1; char *nptr; // *ptr => ‘h’ В результате, в памяти будет следующая последовательность байтов:

Теперь прибавим к первому указателю 4 и запишем результат во второй указатель: nptr = ptr+4; // *nptr => ‘o’ То есть второй указатель теперь указывает на последний из символов. Также справедливо следующее: // nptr-ptr => 4; // *(ptr+1) => ‘e’ // *(ptr+2) => ‘l’ // *(ptr+3) => ‘l’ // *(ptr+4) => ‘o’ // *(ptr+5) => 0 То есть указатель плюс число есть новый указатель, у которого

потом с помощью символа * можно получить значение переменной по новому адресу. Тонкий момент, который здесь нужно учесть заключается в следующем.
Если к указателю ptr прибавить число i, то в результате получится адрес, смещенный не на i байтов, а на i*sizeof(type) байтов, где type – это тип переменной, на которую указывает указатель. То есть отсчет идет не в байтах, а как бы в единицах того типа, на который указывает указатель. Если ptr есть указатель на char, то так как char занимает в памяти один байт, ptr+1 указывает на область памяти, отстоящую от ptr действительно на 1 байт. А если ptr есть указатель на int, то ptr+1 указывает на область памяти, отстоящую от ptr на два байта, потому что переменная типа int занимает в памяти 2 байта. Аналогично, разность ptr2-ptr1 даст не количество байтов между областями памяти, на которые указывают ptr1 и ptr2, а количество переменных, которые могут «уместиться» в памяти между ptr1 и ptr2. Причем переменных именно того типа, указателями на который ptr1 и ptr2 являются. Массивы Волей разработчиков языка, массивы тесно связаны с указателями. Объявим, например, массив символов и запишем в него строчку (которая, как вы помните, обязательно неявно заканчивается символом с кодом 0): char s[6] = ”hello”; // s[0] => ‘h’ // s[1] => ‘e’ // s[2] => ‘l’ // s[3] => ‘l’ // s[4] => ‘o’ // s[5] => 0 char *ptr; ptr = s;
Поэтому к элементам массива можно обращаться как с помощью обычного синтаксиса имя[индекс], так и используя возможности указателей: s[0] => ‘h’ // *s => ‘h’ // *ptr => ‘h’ s[1] => ‘e’ // *(s+1) => ‘e’ // *(ptr+1) => ‘e’ Прибавив к любому из указателей, например, 1, мы заставим этот указатель указывать на следующий символ массива: // *ptr => ‘h’ ptr++; // *ptr => ‘e’ Обратите внимание, что *(s+1) и (*s)+1 – это совсем разные вещи. Почему так? В первом случае сначала к указателю s будет прибавлена единица, так что s+1 – это указатель на следующий символ за тем, на который указывает s. Следовательно *(s+1) – это
сам следующий символ, то есть ‘e’. Во втором случае, сначала выполнится операция *, а стало быть, *s – это символ ‘h’. А после выполнения операции *, к результату ее будет прибавлена единица. Следовательно символ ‘h’ будет расценен как число 104, и результатом (*s)+1 будет число 105, или символ ‘i': char c; c = *(s+1) // c => ‘e’ c = (*s)+1 // c => ‘i’ Динамические массивы Динамическим называется массив, размер которого может меняться во время исполнения программы. Для изменения размера динамического массива язык программирования, поддерживающий такие массивы, должен предоставлять встроенную функцию или оператор. Динамические массивы дают возможность более гибкой работы с данными, так как позволяют не прогнозировать хранимые объёмы данных, а регулировать размер массива в соответствии с реально необходимыми объёмами. При объявлении массива необходимо задать точное количество элементов. На основе этой информации при запуске программы автоматически выделяется необходимый объем памяти. Иными словами, размер массива необходимо знать до выполнения программы. Во время выполнения программы увеличить размер существующего массива нельзя. Чтобы произвести увеличение массива во время выполнения программы необходимо выделить достаточный объем памяти с помощью оператора new, перенести существующие элементы, а лишь затем добавить новые элементы. Управление динамической памятью полностью лежит на плечах программиста, поэтому после завершения работы с памятью необходимо самим возвратить память операционной системе с помощью оператора delete. Если память не возвратить операционной системе, то участок памяти станет недоступным для дальнейшего использования. Подобные ситуации приводят к утечке памяти. Сделать эти участи опять доступными можно только после перезагрузки компьютера. Выделение памяти под массив производится следующим образом: <Указатель> = new <Тип данных>[<Количество элементов>]; Освободить выделенную память можно так: delete [] <Указатель>; Пример объявления динамического массива на языках C/C++ Одномерный динамический массив: Создаем массив с 10-ю элементами типа int: int *mas = new int[10]; Получить доступ к значению каждого элемента можно по индексу (порядковый номер): mas[0] = 2; // присвоили значение 2 нулевому элементу массива mas mas[1] = 7; // присвоили

значение 7 первому элементу массива mas //... и т.д. Следовательно, если брать такой подход, то вам понадобится около десяти строк кода, чтобы проинициализировать весь массив. Для того, чтобы этого избежать напишем тоже самое в цикле: for(int i = 0; i < 10; i++){ cin>>mas[i]; // пользователь вводит значение каждого i-того элемента массива } После чего работаем с массивом. Также его можно вывести на экран:
for(int i = 0; i < 10; i++){ cout << mas[i] << endl; } Для освобождения из памяти одномерного динамического массива используем оператор delete: delete []mas; В случае, если массиву память будет выделена при помощи malloc, то для освобождения такого массива из памяти следует использовать функцию free(mas); Дружественные (friend) функции Object 1 Мы уже познакомились с основным правилом ООП - данные(внутренние переменные) объекта защищены от воздействий из вне и доступ к ним можно получить только с помощью методов(функций) объекта. Но бывают такие случаи, когда нам необходимо получить доступ к данным объекта не используя его интерфейс. Зачем это может понадобиться ? Как я уже как-то упоминал, при доступе к внутренним переменным объекта через его методы уменьшается эффективность работы за счет затрат на вызов метода. В большинстве случаев нам это не критично, но не всегда. В некоторых случаях это может играть существенную роль. Конечно, можно добавить новый метод к классу для получения прямого доступа к внутренним переменным. Однако, в большинстве случаев, интерфейс объекта (методы) спланирован для выполнения определенного круга операций, и наша функция может оказаться как бы ни к месту. А если хуже того, нам необходимо получить прямой доступ к внутренним данным двух разных объектов ? Возникает проблема. Именно для решения подобных задач и существует возможность описание функции, метода другого класса или даже класса как дружественного(friend). Итак, как же работает этот механизм. Для описания дружественной тому или иному классу функции(метода или класса) необходимо в описании этого класса объявить (описать) дружественную функцию с указанием ключевого слова friend. Если функция дружественна нескольким классам, то надо добавить это описание во все классы, к внутренним данным которых производим обращение. В большинстве языков ООП не имеет различия в какой раздел описания класса(publuc, protected или private) вставлено описание дружественной функции. // Описание класса A class A { //... void z(); // Описание функции z класса A };
// Описание класса B class B { //... friend void A::z(); // Описание функции z класса A как дружественной // классу B, т.е. из функции z класса A можно // получить доступ к внутренним переменным класса B }; // Описание класса C class C { //... friend class A; // Описание класса A как дружественного классу C, // все функции класса A будут дружественны классу C и // из любой функции класса A можно получить доступ к // внутренним переменным класса C }
Лекция 5. Принципы ООП. Наследование. Полиморфизм. Наследование и зоны видимости. (см. слайды П_1_5_Наследование)
Ключевые черты ООП хорошо известны: • Первая — инкапсуляция — это определение классов — пользовательских типов данных, объединяющих своё содержимое в единый тип и реализующих некоторые операции или методы над ним. Классы обычно являются основой модульности, инкапсуляции и абстракции данных в языках ООП. • Вторая ключевая черта, — наследование — способ определения нового типа, когда новый тип наследует элементы (свойства и методы) существующего, модифицируя или расширяя их. Это способствует выражению специализации и генерализации. • Третья черта, известная как полиморфизм, позволяет единообразно ссылаться на объекты различных классов (обычно внутри некоторой иерархии). Это делает классы ещё удобнее и облегчает расширение и поддержку программ, основанных на них. Инкапсуляция, наследование и полиморфизм — фундаментальные свойства, которыми должен обладать язык, претендующий называться объектно-ориентированным (языки, не имеющие наследования и полиморфизма, но имеющие только

классы, обычно называются основанными на классах). Наследование Наследование (inheritance) - это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов (hierarchical classification). Применение иерархии классов делает управляемыми большие потоки информации. Например, подумайте об описании жилого дома. Дом - это часть общего класса, называемого строением. С другой стороны, строение - это часть более общего класса - конструкции, который является частью ещё более общего класса объектов, который можно назвать созданием рук человека. В каждом случае порождённый класс наследует все, связанные с родителем, качества и добавляет к ним свои собственные определяющие характеристики. Без использования иерархии классов, для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающи его определяли. Однако при использовании наследования можно описать объект путём определения того общего класса (или классов), к которому он относится, с теми специальными чертами, которые делают объект уникальным. Наследование играет очень важную роль в OOP. class имя_производного: имя_базового_класса {}; C++ позволяет наследоваться по разным зонам видимости: class B : public A{ //public наследование }; class C : protected A{ //protected наследование }; class Z : private A{ //private наследование }; Наследование и модификаторы доступа Несколько приёмов, с помощью которых можно “достучаться” до закрытых функций или данных. Допустим, у нас есть класс some и нам нужно обнулить закрытую переменнуюс: Модифицировать определение класса, добавив друга (функцию или класс) 1 class some { 2 friend class some_friend;
3 public: 4 int a; 5 protected: 6 int b; 7 private: 8 int c; 9 }; 10 11 class some_friend { 12 public: 13 static void hack(some& obj) { 14 obj.c = 0; 15 } 16 }; Воспользоваться препроцессором: 1 #define private public 2 3 class some { 4 public: 5 int a; 6 protected: 7 int b; 8 private: 9 int c; 10 }; 11 12 void hack(some& obj) { 13 obj.c = 0; 14 } Создать класс с таким же расположением в памяти и воспользоваться reinterpret_cast для преобразования указателей: 1 class some { 2 public: 3 int a; 4 protected:
5 int b; 6 private: 7 int c; 8 }; 9 10 class hack_some { 11 public: 12 int a; 13 int b; 14 int c; 15 }; 16 17 void h(some& obj) { 18 reinterpret_cast<hack_some*>(&obj)->c = 0; 19 } Если у “взламываемого” класса есть шаблонная функция, можно её специализировать своим типом: 1 class some { 2 public: 3 int a; 4 template<class T> void func(void) { 5 a = b + c; 6 } 7 protected: 8 int b; 9 private: 10 int c; 11 }; 12 13 class hack_template_param{}; 14 15 template<> 16 void some::func<hack_template_param>(void) { 17 c = 0; 18 }
19 20 void hack(void) { 21 some o; 22 o.func<hack_template_param>(); 23 }; Очевидно, что способ с reinterpret_cast работает только для доступа к закрытым членам данных или вызова виртуальных функций. Остальные же способы позволяют как модифицировать закрытые данные, так и вызывать закрытые невиртуальные методы. Полиморфизм Полиморфизм (polymorphism) (от греческого polymorphos) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Например для языка Си, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трёх различных функций: abs(), labs() и fabs(). Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В C++ каждая из этих функций может быть названа abs(). Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. В C++ можно использовать одно имя функции для множества различных действий. Это называется перегрузкой функций (function overloading). В более

общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает мнижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор. Вам, как программисту, не нужно делать этот выбор самому. Нужно только помнить и использовать общий интерфейс. Пример из предыдущего абзаца показывает, как, имея три имени для функции определения абсолютной величины числа вместо одного, обычная задача становится более сложной, чем это действительно необходимо. Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в Си, символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. В C++ вы можете применить эту концепцию и к другим, заданным вами, типам данных. Такой тип полиморфизма называется перегрузкой операторов (operator overloading). Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путём создания общего для них стандартного интерфейса для реализации похожих действий. Полиморфизм — возможность объектов с одинаковой спецификацией иметь различную реализацию. Полиморфизм реализуется с помощью наследования классов и виртуальных функций. Класс-потомок наследует сигнатуры методов класса-родителя, а реализация, в результате переопределения метода, этих методов может быть другой, соответствующей специфике класса-потомка.

Лекция 6. Раннее и позднее связывание. Виртуальные методы. Таблица виртуальных функций. Абстрактные классы. Чистые виртуальные функции. (см. слайды П_1_6_виртуальные_методы)

В C++ виртуальные функции (virtual functions) позволяют использовать полиморфизм (polymorhpism) классов. Так как виртуальные функции могут использоваться только внутри классов, то иногда их называют виртуальными методами (virtual methods). Прежде чем воспользоваться виртуальными методами, мы рассмотрим работу обычных методов класса. Статическое или раннее связывание (static/early binding) Давайте разберёмся, как происходит вызов обычных функций и методов классов. Вызов обычных функций и методов происходит через механизм, называемый статическим (статичным) связыванием (static binding) или ранним связыванием (early binding). Раннее связывание использовалось во всех функциях и методах наших программ за исключением тех случаев, где мы использовали указатели на функции. Когда мы запускаем сборку (building) программы, компилятор просматривает исходный код и превращает все операторы в команды процессора. Допустим, в коде встречается вызов какой-нибудь функции: someFunction(arg); Если это обычная функция (не указатель на функцию), то при вызове используется механизм раннего связывания. Во время компиляции для кода (определения) функции выделяется память, и назначаются адреса для каждого оператора. Первый адрес в определении (теле функции) является адресом функции. При вызове someFunction, процессор будет переходить на адрес функции и начнёт выполнять тело функции. Самое важное здесь то, что адрес функции назначается во время компиляции, и именно этот адрес используется при вызове функции. Это и есть раннее или статичное связывание. Т.е. имя функции крепко привязано к адресу функции. Теперь взглянем на небольшой пример: class Base { public: void Method () {   cout << "Базовый класс\n"; } };
class Derived : public Base {};
// внутри main Base b; Derived d; b.Method(); d.Method();
//-------- Вывод: Базовый класс Базовый класс

На экран будет выведено две строки Базовый класс. На этапе компиляции память выделяется для двух копий Method - для базового класса и для производного. Оба адреса привязываются

к именам методов: Base::Method, Derived::Method. Т.е. когда в коде мы вызываем Method, то вызывается метод, соответствующий типу объекта. Чтобы увидеть, что для каждого объекта вызывается свой метод, давайте переопределим метод Derived::Method: void Method () { cout << "Производный класс\n"; }
// внутри main Base b; Derived d; b.Method(); d.Method();
//-------- Вывод: Базовый класс Производный класс Здесь хорошо видно, что вызываются два разных метода. Теперь следующий пример. Определения классов оставим без изменений. Поработаем с указателями: Base* b = new Derived; Derived* d = new Derived; b->Method(); d->Method();
//-------- Вывод: Базовый класс Производный класс Самое важное здесь то, что компилятор спокойно "проглатывает" тот факт, что указатель на Base указывает на производный класс. Дело в том, что базовый и производный классы являются совместимыми по типу. Во время выполнения программы процессор видит, что b - это указатель на Base. Процессор не обращает внимание, что на самом деле этот указатель указывает на объект Derived. При вызове метода объекта bпроцессор переходит к адресу Base::Method. Полиморфизм (polymorphism) и полиморфные типы (polymorphic types) Рассмотрим гипотетическую ситуацию: в игре есть несколько типов монстров. Все монстры могут атаковать (attack) и перемещаться (move). При этом каждый вид монстров делает это по своему. Неплохо было бы иметь возможность хранить объекты всех этих классов вместе и использовать одинаковый синтаксис для вызова методов этих классов. Это и есть полиморфизм (polymorphism) - много (от греческого поли) форм (от греческого морф). Т.е. объекты этих классов должны храниться в одном массиве. Понятно, что для этого не годится массив объектов, так как в таком массиве для элементов выделяется фиксированное количество памяти. Соответственно, для поддержки полиморфизма нужно использовать массив указателей. Какой тип указателей выбрать? Как мы выяснили раньше, в указателе на базовый тип можно хранить объект любого производного типа - базовый и производный классы являются совместимыми по типу. Почему используется указатели именно на базовый тип? Потому что это более общий класс и от него наследуют все производные классы. Классы, используемые для получения эффекта полиморфизма, называют полиморфными типами (polymorphic types). Позднее/динамическое связывание (late/dynamic binding) Поздним связыванием в C++ обладают указатели на функции (function pointers). Мы их уже разбирали, поэтому сложностей возникнуть не должно. Сразу пример: int someFunction (int arg); int (*functionPointer)(int arg); functionPointer = someFunction. someFunction обладает ранним связыванием. Т.е. на этапе компиляции для этой функции выделяется участок памяти, а первый адрес этого участка становится адресом функции. Адрес функции жёстко привязан к имени функции - их нельзя отделить. functionPointer обладает динамическим (dynamic) или поздним связыванием (late binding). На какую функцию указывает этот указатель, становится известно только во время выполнения программы.При этом functionPointer может указывать на любую функцию, т.е. значение указателя functionPointer может меняться во время выполнения программы. Это и есть позднее связывание. Виртуальные функции/методы (virtual functions/methods) Чтобы объявить функцию как виртуальную, необходимо добавить ключевое слово virutal перед именем возвращаемого типа: class Base { public: virtual void Method () { cout << "Базовый класс\n"; } }; Вносить изменения в производные классы не нужно. Хотя можно и там добавить ключевое слово virtual (это не обязательно). Теперь посмотрим на наш код: Base* b = new Derived; Derived* d = new Derived; b->Method(); d->Method();
//-------- Вывод: Производный класс Производный класс Наконец-то мы можем создать массив указателей на базовый класс и размещать там объекты любого производного класса: BaseMonster* monsters[3]; monsters[0] = new MonsterA;
monsters[1] = new MonsterB; monsters[2] = new MonsterC;
for (int i=0;i<3;++i) monsters[i]->attack(); Несколько замечаний по виртуальным функциям: 1.

Виртуальные функции используются только в классах. Поэтому часто используется название - виртуальные методы. 2. В массивах указателей на базовый класс можно хранить объекты только полиморфных типов (базовый и все производные). 3. В массив нужно объединять только те объекты, которые обладают методами с одинаковыми названиями, но разной реализацией. Таблица виртуальных функций (virtual function table) Для виртуальных методов память выделяется точно так же, как и для обычных: на этапе компиляции под эти методы выделяются участки памяти, первые адреса которых являются адресами методов. Но так как методы виртуальные, то фактические адреса метода не привязывается к именам: Base::vf и Derived::vf. Адрес метода, который назначается на этапе компиляции при выделении памяти, будем называть настоящим (или фактическим) адресом. Когда в базовом классе объявляется хотя бы одна виртуальная функция, то для всех полиморфных классов создаётся таблица виртуальных функций (virtual function table). Таблица виртуальных функций - это одномерный массив указателей на функции. Количество элементов в массиве равно количеству виртуальных функций в классе. Для каждого полиморфного класса (базового и всех производных) создаётся своя таблица виртуальных методов. Количество элементов во всех этих таблицах одинаковое. Именно в таблице виртуальных функций записываются настоящие адреса методов, т.е. элемент таблицы является указателем на функцию. Для всех полиморфных классов таблицы виртуальных функций будут содержать разные значения. Для каждого класса здесь будут записаны адреса методов данного класса. Абстрактные классы (abstract classes) и чистые виртуальные функции (pure virtual functions) Очень часто в программах не требуется создавать объекты базовых классов. Т.е. базовые классы нужны только для того, чтобы построить иерархию классов и определить общие свойства для производных классов. Такие классы можно сделать абстрактными (abstract class). При попытке создания объекта абстрактного класса, компилятор выдаст ошибку. Чтобы сделать класс абстрактным, нужно объявить одну из виртуальных функций чистой. Чистая виртуальная функция (pure virtual function) как бы намекает, что она будет реализована в производных классах. Чтобы сделать виртуальную функцию чистой (pure), нужно добавить после заголовка функции символы =0 (знак равенства и ноль): class Base { public:
virtual void method () =0; virtual ~Base() =0; }; В данном случае уже не нужно писать определение такой функции. Помимо этого теперь нельзя создавать объекты класса Base, так как он стал абстрактным. Символы =0 необязательно добавлять ко всем виртуальным функциям, достаточно добавить к одной.

Лекция 7. Перегрузка операций. Файловый ввод-вывод. Понятие шаблона. (см. слайды П_1_7_Перегрузка_операций)

Перегрузка операторов — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются. Синтаксис перегрузки операторов очень похож на определение функции с именем operator@, где @ — это идентификатор оператора (например +, -, <<, >>). Рассмотрим простейший пример: class Integer { private: int value; public: Integer(int i): value(i) {} const Integer operator+(const Integer& rv) const { return (value + rv.value); } }; В данном случае, оператор оформлен как член класса, аргумент определяет значение, находящееся в правой части оператора. Вообще, существует два основных способа перегрузки операторов: глобальные функции, дружественные для класса, или подставляемые функции самого класса. В большинстве случаев, операторы (кроме условных) возвращают объект, или ссылку на тип, к которому относятся его аргументы (если типы разные, то вы сами решаете как интерпретировать результат вычисления оператора). Рассмотрим примеры перегрузки унарных операторов для определенного выше класса Integer. Заодно определим их в виде дружественных функций и рассмотрим операторы декремента и инкремента: class Integer { private: int value; public: Integer(int i): value(i) {}

```
    //унарный +    friend const Integer& operator+(const Integer& i);
    //унарный     friend const Integer operator-(const Integer& i);
    //префиксный инкремент    friend const Integer& operator++(Integer& i);
    //постфиксный инкремент    friend const Integer operator++(Integer& i, int);
    //префиксный декремент    friend const Integer& operator--(Integer& i);
    //постфиксный декремент    friend const Integer operator--(Integer& i, int); };
//унарный плюс ничего не делает. const Integer& operator+(const Integer& i) {    return i.value; }
const Integer operator-(const Integer& i) {    return Integer(-i.value); }
//префиксная версия возвращает значение после инкремента const Integer& operator++(Integer& i) {    i.value++;    return i; }
//постфиксная версия возвращает значение до инкремента const Integer operator++(Integer& i, int) {    Integer oldValue(i.value);    i.value++;    return oldValue; }
//префиксная версия возвращает значение после декремента const Integer& operator--(Integer& i) {    i.value--;    return i; }
//постфиксная версия возвращает значение до декремента const Integer operator--(Integer& i, int) {    Integer oldValue(i.value);    i.value--;    return oldValue; }
```

Теперь вы знаете, как компилятор различает префиксные и постфиксные версии декремента и инкремента. В случае, когда он видит выражение ++i, то вызывается функция operator++(a). Если же он видит i++, то вызывается operator++(a, int). То есть вызывается перегруженная функция operator++, и именно для этого используется фиктивный параметр int в постфиксной версии. Рассмотрим синтаксис перегрузки бинарных операторов. Перегрузим один оператор, который возвращает l-значение, один условный оператор и один оператор, создающий новое значение (определим их глобально):

```
class Integer { private:    int value; public:    Integer(int i): value(i)    {}    friend const Integer operator+(const Integer& left, const Integer& right);
    friend Integer& operator+=(Integer& left, const Integer& right);
    friend bool operator==(const Integer& left, const Integer& right); };
const Integer operator+(const Integer& left, const Integer& right) {     return Integer(left.value + right.value); }
Integer& operator+=(Integer& left, const Integer& right) {    left.value += right.value;    return left; }
bool operator==(const Integer& left, const Integer& right) {    return left.value == right.value; }
```

Во всех этих примерах операторы перегружаются для одного типа, однако, это необязательно. Можно, к примеру, перегрузить сложение нашего типа Integer и определенного по его подобию Float. Как можно было заметить, в примерах используются различные способы передачи аргументов в функции и возвращения значений операторов. • Если аргумент не изменяется оператором, в случае, например унарного плюса, его нужно передавать как ссылку на константу. Вообще, это справедливо для почти всех арифметических операторов (сложение, вычитание, умножение...) • Тип возвращаемого значения зависит от сути оператора. Если оператор должен возвращать новое значение, то необходимо создавать новый объект (как в случае бинарного плюса). Если вы хотите запретить изменение объекта как l-value, то нужно возвращать его константным. • Для операторов присваивания необходимо возвращать ссылку на измененный элемент. Также, если вы хотите использовать оператор присваивания в конструкциях вида (x=y).f(), где функция f() вызывается для для переменной x, после присваивания ей y, то не возвращайте ссылку на константу, возвращайте просто ссылку. • Логические операторы должны возвращать в худшем случае int, а в лучшем bool. Некоторые операторы в C++ не перегружаются в принципе. По всей видимости, это сделано из соображений безопасности. • Оператор выбора члена класса ".". • Оператор разыменования указателя на член класса ".*" • В C++ отсутствует оператор возведения в степень (как в Fortran) "**".

• Запрещено определять свои операторы (возможны проблемы с определением приоритетов). • Нельзя изменять приоритеты операторов Как мы уже выяснили, существует два способа операторов — в виде функции класса и в виде дружественной глобальной функции. Роб Мюррей, в своей книге C++ Strategies and Tactics определил следующие рекомендации по выбору формы оператора: Оператор Рекомендуемая форма Все унарные операторы Член класса = () [] -> ->* Обязательно член класса += -= /= *= ^= &= |= %= >>= <<= Член класса Остальные бинарные операторы Не член класса Файловый ввод/вывод Все операции, применимые в стандартному вводу и выводу, могут быть также применены к файлам. Чтобы использовать файл для ввода или вывода, мы должны включить еще один заголовочный файл: #include <fstream> Перед тем как открыть файл для вывода, необходимо объявить объект типа ofstream: ofstream outfile("name-of-file"); Проверить, удалось ли нам открыть файл, можно следующим образом: if (! outfile) // false, если файл не открыт cerr << "Ошибка открытия файла.\n" Так же открывается файл и для ввода, только он имеет тип ifstream: ifstream infile("name-of-file"); if (! infile) // false, если файл не открыт cerr << "Ошибка открытия файла.\n" Ниже приводится текст простой программы, которая читает файл с именем in_file и выводит все прочитанные из этого файла слова, разделяя их пробелом, в другой файл, названный out_file. #include <iostream> #include <fstream> #include <string> int main() { ifstream infile("in_file"); ofstream outfile("out_file");
if (! infile) { cerr << "Ошибка открытия входного файла.\n" return -1; }
if (! outfile) { cerr << "Ошибка открытия выходного файла.\n" return -2; } string word; while (infile >> word) outfile << word << ' '; return 0; }
Шаблоны Шаблоны (англ. template) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию). В C++ возможно создание шаблонов функций и классов. Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем). Шаблон функции начинается с ключевого слова template, за которым в угловых скобках следует список параметров. Затем следует объявление функции: template< typename T > void sort(T array[], int size); // прототип: шаблон sort объявлен, но не определён

template< typename T > void sort(T array[], int size) // объявление и определение { T t; for (int i = 0; i < size - 1; i++) for (int j = size - 1; j > i; j--) if (array[j] < array[j-1]) { t = array[j]; array[j] = array[j-1]; array[j-1] = t; } }

template< int BufferSize > // целочисленный параметр char* read() { char *Buffer = new char[BufferSize]; /* считывание данных */ return Buffer; } Ключевое слово typename появилось сравнительно недавно, поэтому стандарт допускает использование class вместо typename: template< class T > Вместо T допустим любой другой идентификатор. Простейшим примером служит определение минимума из двух величин. Если a меньше b то вернуть a, иначе - вернуть b В отсутствие шаблонов программисту приходится писать отдельные функции для каждого используемого типа данных. Хотя многие языки программирования определяют встроенную функцию минимума для элементарных типов (таких как целые и вещественные числа), такая функция может понадобится и для сложных (например «время» или «строка») и очень сложных («игрок» в онлайн-игре) объектов. Так выглядит шаблон функции определения минимума: template< typename T > T min(T a, T b) { return a < b ? a : b; } Для вызова этой функции можно просто использовать её имя:
min(1, 2); min('a', 'b'); min(string("abc"), string("cde")); Вообще говоря, для вызова шаблонной функции, необходимо указать значения для всех параметров шаблона. Для этого после имени шаблона указывается список значений в угловых скобках: int i[5] = { 5, 4, 3, 2, 1

}; sort< int >(i, 5);

char c[] = "бвгда"; sort< char >(c, strlen(c));

sort< int >(c, 5); // ошибка: у sort< int > параметр int[] а не char[]

char *ReadString = read< 20 >(); delete [] ReadString; ReadString = read< 30 >(); Для каждого набора параметров компилятор генерирует новый экземпляр функции. Процесс создания нового экземпляра называется инстанцированием шаблона. В примере выше компилятор создал две специализации шаблона функции sort (для типов char и int) и две — шаблона read (для значений BufferSize 20 и 30). Последнее скорее всего расточительно, так как для каждого возможного значения параметра компилятор будет создавать новые и новые экземпляры функций, которые будут отличаться лишь одной константой.
Лекция 8. Шаблонные классы. (см. слайды П_1_7_Шаблонные_классы)
Шаблоны классов В классе, реализующем связный список целых чисел, алгоритмы добавления нового элемента списка, поиска нужного элемента не зависят от того, что элементы списка — целые числа. Те же алгоритмы применялись бы и для списка символов, строк, дат, классов игроков, и так далее. template< class T > class List { /* ... */ public: void Add(const T& Element); bool Find(const T& Element); /* ... */ }; Для использования шаблона класса, необходимо указать его параметры: List<int> li; List<string> ls; li.Add(17); ls.Add("Hello!"); Параметрами шаблонов могут быть: параметры-типы, параметры обычных типов, параметры-шаблоны. Для параметров любого типа можно указывать значения по умолчанию. template< class T1, // параметр-тип typename T2, // параметр-тип int I, // параметр обычного типа T1 DefaultValue, // параметр обычного типа template< class > class T3, // параметр-шаблон class Character = char // параметр по умолчанию > Если в шаблоне класса или функции необходимо использовать один и тот же шаблон, но с разными параметрами, то используются параметры-шаблоны. Например: template< class Type, template< class > class Container > class CrossReferences { Container< Type > mems; Container< Type* > refs; /* ... */ };

CrossReferences< Date, vector > cr1; CrossReferences< string, set > cr2; Нельзя использовать шаблоны функций в качестве параметров-шаблонов. Члены шаблонов классов Члены шаблона класса являются шаблонами, причём с той же, что и у шаблона класса, параметризацией. В частности это означает, что определение функций-членов следует начинать с заголовка шаблона: template< class T > class A
{ void f(T data); void g(void); public: A(); };

template< class T > void A<T>::f(T data);

template< class T > void A<T>::g(void); Внутри области видимости шаблона не нужно повторять спецификатор. Это значит, что например A<T>::A() — это конструктор, хотя можно писать и A<T>::A<T>(). Если параметром шаблона является класс, у которого есть член, являющийся типом данных, то для использования этого члена, нужно применять ключевое слово typename. Например: class Container { public: int array[15]; typedef int* iterator; /* ... */ iterator begin() { return array; } };

template< class C > void f(C& vector) { C::iterator i = vector.begin(); // ошибка typename C::iterator i = vector.begin(); } Проблемы возникают и с членами-шаблонами. Если шаблон, который является членом класса, который в свою очередь является параметром шаблона, используется в этом шаблоне и не допускает выведения параметров, то необходимо

использовать квалификатор template: class A { /* ... */ public: template< class T > T& ConvertTo(); template< class T > void ConvertFrom(const T& data); };

template< class T > void f(T Container) { int i1 = Container.template ConvertTo<int>() + 1; Container.ConvertFrom(i1); // квалификатор не нужен }

Лекция 9. Модели компиляции шаблонов. Специализация шаблонов. (см. слайды П_1_9_Модели_компиляции)

Модели компиляции Программа на Си++ представляет собой совокупность единиц трансляции (translation-units), каждая из которых может храниться и обрабатываться компилятором независимо от других. Кодовые образы единиц трансляции, по существу, представляют собой заготовки, подлежащие последующему объединению. Формирование исполняемого образа всей программы осуществляется отдельным, независимым от компилятора процессором, который традиционно называется редактором связей. Такой подход, который естественно именовать раздельной компиляцией, сам по себе не может вызывать возражений. Удачное и обоснованное разбиение (проекта, а в дальнейшем и текста) программы на отдельные компоненты дает возможность повысить ее надежность и организовать коллективную разработку. Кроме того, раздельная компиляция позволяет создавать многократно используемые компоненты. Согласно правилу одного описания (one definition rule), указанные сущности могут встречаться в программе не более одного раза. Однако, чтобы контролировать выполнение этого правила компилятор вынужден был бы анализировать все единицы трансляции, образующие программу, что вступает в противоречие с принципом раздельной компиляции в том виде, в котором этот принцип заимствован из языка Си. Поэтому разработчики Стандарта вынуждены были сопроводить правило одного описания оговоркой, что при его нарушении "диагностика не требуется". Такое допущение, по существу, отменяет правило одного описания, так как на практике оно означает, что в нескольких единицах трансляции можно задать одноименные объекты; такая ошибка диагностирована не будет, а эффект использования таких объектов в пределах всей программы никак не определяется языком и его компилятором и полностью зависит от "интеллекта" и особенностей реализации редактора связей. Как частичная компенсация указанного недостатка, для организации межмодульного взаимодействия на практике используются так называемые заголовочные файлы. Для каждой единицы трансляции создается специальный текст ("интерфейс"), в который помещаются объявления сущностей из этой единицы трансляции, имеющих внешнее связывание. Такой текст посредством механизма препроцессирования включается в другие единицы трансляции, использующие эти сущности. Этот подход нельзя считать полноценным решением; он, скорее является некоторым суррогатом модульности. Рассмотрим данное обстоятельство на простом примере. Пусть имеется описание функции: void f (char* t) { ... } и пусть необходимо необходимо сделать эту функцию доступной вне той единицы трансляции, в которой она описана; такая возможность обеспечивает связь компонент по управлению и является типичной в реальном программировании. Стандартное решение для обычных функций выглядит следующим образом. Описание функции располагается в одной единице трансляции, а ее объявление помещается в заголовочный файл, который посредством директивы #include вводится во все единицы трансляции, использующие эту функцию. Теперь рассмотрим такую же ситуацию для шаблона функции: template<class T> void f (T t) { . . . } Если использование такого шаблона предполагается в нескольких единицах трансляции, то прием, описанный для функций, не даст желаемого результата. Прямое решение этой проблемы достаточно очевидно: вынести в заголовочный файл полное описание шаблона и в таком виде включать его во все использующие этот шаблон

единицы компиляции. Именно такой подход (назовем его "включающим") широко используется на практике, формируя негативную тенденцию "разбухания" заголовочных

файлов: по некоторым оценкам, активное использование механизма шаблонов приводит к тому, что до 40 процентов программного текста сосредотачивается в заголовках. В процессе стандартизации языка Си++ проблема "встраивания" механизма шаблонов в принцип раздельной компиляции являлась предметов продолжительных и острых дискуссий. Во-первых, "включающий" принцип был признан допустимым; с этой целью правило одного описания было соответствующим образом скорректировано (помимо классов и перечислимых типов, в различных единицах компиляции теперь могут входить идентичные "шаблонные" описания). Тем самым, была фактически легализована повсеместная практика переноса большой части программного текста в заголовочные файлы. Во-вторых, в механизм шаблонов были внесены усовершенствования, направленные на поддержку раздельной компиляции объявлений и описаний шаблонов (такая возможность, в противоположность "включающему", иногда называется "раздельным" подходом). Основная идея заключается во введении служебного слова export, которое может использоваться, чтобы сделать описание шаблона доступным вне той единицы трансляции, в которой оно появляется. В общих чертах механизм "экспортирования" выглядит следующим образом. Стандарт устанавливает, что если некоторое объявление шаблона предваряется служебным словом export, то этого достаточно для обработки той единицы трансляции, в которой это объявление встречается. Например, в следующем фрагменте export template <class T> void f (T); . . . f(1); . . . f('a'); . . . наличие слова export перед объявлением шаблона делает возможным трансляцию данной единицы, включая обработку настроек данного шаблона. Так как для таких настроек требуется (полное) описание шаблона f, компилятор должен каким-либо образом получить его. Конкретные способы поиска в стандарте не фиксируются. В результате стандарт поддерживает обе формы организации исходных текстов - "раздельную" и "включающую". Специализация шаблонов Например, простейший шаблонный класс 'A' можно задать вот так:

Через некоторое время мы захотим, чтобы наш класс для всех типов работал одинаково, а для какого-нибудь хитрого вроде int — по-другому. Фигня вопрос, пишем специализацию: выглядит так же как объявление но параметры шаблона в угловых скобках не указываем, вместо этого указываем конкретные аргументы шаблона после его имени: template<> class A< int > {}; // здесь int - это аргумент шаблона Готово, можно писать методы и поля специальной реализации для int. Такая специализация обычно называется полной (full specialization или explicit specialization). Если внимательно читать ISO стандарт C++, то можно обнаружить интересное утверждение: создав специализированный шаблонный класс мы создаем новый шаблонный
класс. Специализированный шаблонный класс может содержать методы, поля или объявления типов которых нет в шаблонном классе который мы специализируем. Удобно, когда нужно чтобы метод шаблонного класса работал только для конкретной специализации — достаточно объявить метод только в этой специализации, остальное сделает компилятор:

То, что специализированный шаблонный класс это совсем-совсем новый и отдельный класс конечно интересно, но магии в этом мало. А магия есть в незначительном следствии — если это отдельный шаблонный класс, то он может иметь отдельные, никак не связанные с неспециализированным шаблонным классом параметры (параметры — это то, что после template в угловых скобках). То есть специализируя шаблонный класс мы можем специализировать его через новые параметры: template< typename S, typename U > class A< std::map< S, U > > {}; И при использовании получившегося шаблонного класса с std::map будет использована специализация, где тип ключа std::map будет доступен как параметр нового шаблона S, а тип значения std::map как U. Такая специализация шаблона, при которой задается новый список параметров и через эти параметры задаются аргументы для специализации называется частичной специализацией (partial specialization).

Лекция 10. Семинар на тему «Шаблоны в C++ - за и против»
Возможные темы докладов: 1. Шаблоны в c++ и модели компиляции 2. Шаблоны с параметрами-шаблонами 3. Метапрограммирование шаблонов 4. Шаблонные методы и шаблонные классы 5. Использование шаблонов и время компиляции 6. Примеры применения шаблонов и так далее.
Лекция 11. Потоки выполнения. Многопоточность. (см. слайды П_1_11_Многопоточность)
Потоки выполнения Поток выполнения — наименьшая единица обработки, исполнение которой может быть назначено операционной системой. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени). Обычно число потоков в системе гораздо больше числа процессоров. Поэтому операционной системе приходится эмулировать многопоточность, заставляя процессор поочередно переключаться между потоками. Время работы процессора разбивается на небольшие интервалы, обычно называемые квантами. Когда квант времени заканчивается, выполнение текущего потока приостанавливается, содержимое регистров процессора сохраняется в специальной области памяти, и он переключается на обработку следующего потока. Когда очередь снова дойдет до этого потока, содержимое регистров будет полностью восстановлено и работа потока продолжится так, как будто она вовсе и не прерывалась. Таким образом, переключение потоков происходит совершенно незаметно для них самих. Многопоточность Многопоточность, как широко распространённая модель программирования и исполнения кода, позволяет нескольким потокам выполняться в рамках одного процесса. Эти потоки выполнения совместно используют ресурсы процесса, но могут работать и самостоятельно. Многопоточная модель программирования предоставляет разработчикам удобную абстракцию параллельного выполнения. К достоинствам многопоточности в программировании можно отнести следующее: • Упрощение программы в некоторых случаях, за счет использования общего адресного пространства. • Меньшие относительно процесса временны‹е затраты на создание потока. • Повышение производительности процесса за счет распараллеливания процессорных вычислений и операций ввода/вывода. Несинхронизированные потоки Первый пример иллюстрирует работу с несинхронизированными потоками. Основной цикл, который является основным потоком процесса, выводит на экран содержимое глобального массива целых чисел. Поток, названный "Thread", непрерывно заполняет глобальный массив целых чисел. #include <process.h> #include <stdio.h> int a[5]; void Thread(void* pParams) { int i, num = 0; while (1) { for (i = 0; i < 5; i++) a[i] = num; num++; } } int main(void) { _beginthread(Thread, 0, NULL); while(1) printf("%d %d %d %d %d\n",
a[0], a[1], a[2], a[3], a[4]); return 0; } Как видно из результата работы процесса, основной поток (сама программа) и поток Thread действительно работают параллельно (красным цветом обозначено состояние, когда основной поток выводит массив во время его заполнения потоком Thread): 81751652 81751652 81751651 81751651 81751651 81751652 81751652 81751651 81751651 81751651 83348630 83348630 83348630 83348629 83348629 83348630 83348630 83348630 83348629 83348629 83348630 83348630 83348630 83348629 83348629 Запустите программу, затем нажмите "Pause" для остановки вывода на дисплей (т. е. приостанавливаются операции ввода/вывода основного потока, но поток Thread продолжает свое выполнение в фоновом режиме) и любую другую клавишу для возобновления выполнения. Критические секции А что делать, если основной поток должен читать данные из массива после его обработки в параллельном процессе? Одно из решений этой проблемы - использование критических секций. Критические секции обеспечивают синхронизацию

подобно мьютексам (о мьютексах см. далее) за исключением того, что объекты, представляющие критические секции, доступны в пределах одного процесса. События, мьютексы и семафоры также можно использовать в "однопроцессном" приложении, однако критические секции обеспечивают более быстрый и более эффективный механизм взаимно-исключающей синхронизации. Подобно мьютексам объект, представляющий критическую секцию, может использоваться только одним потоком в данный момент времени, что делает их крайне полезными при разграничении доступа к общим ресурсам. Трудно предположить что-нибудь о порядке, в котором потоки будут получать доступ к ресурсу, можно сказать лишь, что система будет "справедлива" ко всем потокам. #include <windows.h> #include <process.h> #include <stdio.h> CRITICAL_SECTION cs; int a[5]; void Thread(void* pParams) { int i, num = 0; while (TRUE) { EnterCriticalSection(&cs); for (i = 0; i < 5; i++) a[i] = num; LeaveCriticalSection(&cs); num++; } } int main(void) { InitializeCriticalSection(&cs); _beginthread(Thread, 0, NULL); while(TRUE) { EnterCriticalSection(&cs); printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]); LeaveCriticalSection(&cs); }
return 0; } Мьютексы (взаимоисключения) Мьютекс (взаимоисключение, mutex) - это объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент времени, отсюда и название таких объектов — одновременный доступ к общему ресурсу исключается. Например, чтобы исключить запись двух потоков в общий участок памяти в одно и то же время, каждый поток ожидает, когда освободится мьютекс, становится его владельцем и только потом пишет чтолибо в этот участок памяти. После всех необходимых действий мьютекс освобождается, предоставляя другим потокам доступ к общему ресурсу. Два (или более) процесса могут создать мьютекс с одним и тем же именем, вызвав метод CreateMutex . Первый процесс действительно создает мьютекс, а следующие процессы получают хэндл уже существующего объекта. Это дает возможность нескольким процессам получить хэндл одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает мьютекс. Если используется такой подход, желательно установить флаг bInitialOwner в FALSE, иначе возникнут определенные трудности при определении действительного создателя мьютекса. Несколько процессов могут получить хэндл одного и того же мьютекса, что делает возможным взаимодействие между процессами. Вы можете использовать следующие механизмы такого подхода: • Дочерний процесс, созданный при помощи функции CreateProcess может наследовать хэндл мьютекса в случае, если при его (мьютекса) создании функией CreateMutex был указан параметр lpMutexAttributes . • Процесс может получить дубликат существующего мьютекса с помощью функции DuplicateHandle . • Процесс может указать имя существующего мьютекса при вызове функций OpenMutex или CreateMutex . Вообще говоря, если вы синхронизируете потоки одного процесса, более эффективным подходом является использование критических секций. #include <windows.h> #include <process.h> #include <stdio.h> HANDLE hMutex; int a[5]; void Thread(void* pParams) { int i, num = 0; while (TRUE) { WaitForSingleObject(hMutex, INFINITE); for (i = 0; i < 5; i++) a[i] = num; ReleaseMutex(hMutex); num++; } } int main(void) { hMutex = CreateMutex(NULL, FALSE, NULL); _beginthread(Thread, 0, NULL); while(TRUE) { WaitForSingleObject(hMutex, INFINITE); printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]); ReleaseMutex(hMutex); }
return 0; } События А что, если мы хотим, чтобы в предыдущем примере второй поток запускался каждый раз после того, как основной поток закончит печать содержимого массива, т.е. значения двух последующих строк будут отличаться строго на 1? Событие - это объект синхронизации, состояние которого может быть установлено в сигнальное путем вызова функций SetEvent или PulseEvent . Существует два типа событий:
События полезны в тех случаях, когда необходимо послать сообщение потоку, сообщающее, что произошло определенное событие. Например, при асинхронных операциях ввода и вывода

из одного устройства, система устанавливает событие в сигнальное состояние когда заканчивается какая-либо из этих операций. Один поток может использовать несколько различных событий в нескольких перекрывающихся операциях, а затем ожидать прихода сигнала от любого из них. Поток может использовать функцию CreateEvent для создания объекта события. Создающий событие поток устанавливает его начальное состояние. В создающем потоке можно указать имя события. Потоки других процессов могут получить доступ к этому событию по имени, указав его в функции OpenEvent . Поток может использовать функцию PulseEvent для установки состояния события в сигнальное и затем сбросить состояние в несигнальное после освобождения соответствующего количества ожидающих потоков. В случае объектов с ручным сбросом освобождаются все ожидающие потоки. В случае объектов с автоматическим сбросом освобождается только единственный поток, даже если этого события ожидают несколько потоков. Если ожидающих потоков нет, PulseEvent просто устанавливает состояние события в несигнальное. #include <windows.h> #include <process.h> #include <stdio.h> HANDLE hEvent1, hEvent2; int a[5]; void Thread(void* pParams) { int i, num = 0; while (TRUE) { WaitForSingleObject(hEvent2, INFINITE); for (i = 0; i < 5; i++) a[i] = num; SetEvent(hEvent1); num++; }
} int main(void) { hEvent1 = CreateEvent(NULL, FALSE, TRUE, NULL); hEvent2 = CreateEvent(NULL, FALSE, FALSE, NULL); _beginthread(Thread, 0, NULL); while(TRUE) { WaitForSingleObject(hEvent1, INFINITE); printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]); SetEvent(hEvent2); } return 0; }
Лекция 12. Строки в c/c++. Команды работы со строковыми типами. (см. слайды П_1_12_Строки)
Строковые переменные Очевидно, что строка – это последовательность символов. В С – как в жизни. Сстрока — это последовательность символов. Как известно, последовательности в С представляются массивами или указателями. Символы Символы сами по себе тоже довольно интересны. Как вы уже знаете, символьная переменная — это переменная типа char, занимающая в памяти 1 байт. В отличие от строк, символ — это встроенный интегральный тип в C/C++, для него допустимы все операции, допустимые для интегральных типов. Существуют символьные литералы, они записываются в одинарных кавычках (прямых апострофах). Пример символьного литерала: char sym; sym='A'; //Символьный литерал. Его значение – код символа A(латинское) в используемой кодировке В вышеприведенном примере значением sym является 65 в кодовой таблице ASCII. В этом случае строка sym=’A’ абсолютно эквивалентна строке sym=65. Однако, в целях улучшения портируемости лучше всегда использовать запись в апострофах — рано или поздно программу может потребоваться скомпилировать на платформе, где у символа A другой код. Для записи символьных литералов типа wchar_t используется запись, аналогичная записи для строковых литералов этого типа: wchar_t sym; sym=L'ab'; //Символьный многобайтовый литерал. Количество символов между апострофами зависит от размера типа wchar_t Существует специальный формат для записи символьных литералов – слеш, за которым идет код символа. Такая форма записи необходима, если мы хотим использовать элемент, не отображающийся в печатный символ, например нуль-терминатор, который представляется так: ’\0’. Создание строк Проиллюстрирую создание строк на фрагментах кода с комментариями. char str1[10]; // Строка - массив из 10 символов. Начальное значение символов не определено. char str2[10]="Hello"; /* Используется инициализация (не присваивание!). В первые 5 символов записывается “Hello”, в 6 – нуль-терминатор, значение трех последних не определено.*/ char str3[10]={'H', 'e', 'l', 'l', 'o', '\0'}; //Эквивалентно предыдущему. char str4[10]="Very long line"; //Ошибка. Массив из 10 элементов нельзя инициировать более длинной последовательностью. char str5[]="Very long line"; /*Компилятор автоматически определяет длину массива (в нашем случае 15) и инициализирует его последовательностью символов. */ char* str6; /*Строка - указатель на символ. В большинстве случаев для ее использования потребуется выделить память.*/

str6=(char*) malloc(10); free(str6); Копирование строк Для копирования строк существуют несколько библиотечных функций, наиболее общеупотребительной из которых является функция
char* strcpy(char* dest, const char* src) При использовании этой функции следует соблюдать осторожность. Опасность заключается в том, что даже если исходная строка окажется больше, чем память, выделенная для второй строки (программистом через malloc или компилятором при использовании массивов), функция strcpy никак про это узнать не сможет и продолжит копирование в невыделенную память. Разумеется, последствия будут катастрофическими. Снизить риск такого развития событий способна функция char* strncpy(char* dest, const char* src, size_t count) Последний параметр – максимальное количество копируемых символов. Таким образом, передавая туда размер приемника, вы гарантируете, что функция никогда не выйдет за пределы выделенной памяти. Сравнение строк Для лексикографического сравнения строк используются функции strcmp и stricmp. Первая сравнивает строки с учетом регистра, вторая – без. Однако, все это относится только к латинице. Если вы хотите сравнивать без учета регистра кириллические строки, придется разобраться с локалями. Прототипы этих функций таковы: int stricmp(const char *string1, const char *string2); int strcmp(const char *string1, const char *string2); Обе функции возвращают число меньшее 0, если первая строка меньше второй, большее нуля если первая строка больше второй и 0, если строки лексикографически равны. Длина строки Для вычисления длины строки используется функция size_t strlen(const char *string); Функция возвращает длину строки, не включая нуль-терминатор. Как всегда, следите за тем, чтобы в выделенной под string памяти все же нашелся такой символ. Преобразования строк Зачастую требуется преобразовать число в строку и наоборот. Есть несколько способов сделать это. Можно использовать функции sprintf и sscanf. Например, так: char str[50]; int i=15; int j; sprintf(str, "%d", i); // Записать в str строковое представление i sscanf(str, "%d", &j); // Записать в j число, содержащееся в строке str sprintf(str, "i=%d and j=%d", i, j); // содержимое str: "i=15 and j=15" Эти функции очень похожи на printf и scanf, за исключением того, что они работают не с консолью, а со строковым буфером. Конкатенация (объединение) строк Для конкатенации следует использовать функции. Есть две специальные функции: char* strcat(char* dest, const char* source) char* strncat(char* dest, const char* source, size_t size) Эти функции добавляют к строке, на которую указывает dest, символы из строки source. Первая версия добавляет все символы до нуль-терминатора, вторая – максимум size символов. Результирующая строка завершается нуль-терминатором. Кроме того, можно воспользоваться общей функцией sprintf так: char str1[]="Hello "; char str2[]="world"; char str3[]="!"; char str4[13]; sprintf(str3, "%s%s%s", str1, str2, str3);
Для использования строковых функций вам потребуется подключить к программе соответствующие стандартные заголовки. Это string.h для всех функций, кроме sprintf и sscanf, определенных в stdio.h и функций преобразования, определенных в stdlib.h. В C++ вместо вышеупомянутых заголовочных файлов следует подключать cstring, cstdio и cstdlib соответственно. Класс string Для его работы необходимо в начале программы подключить заголовочный файл string: #include <string> В отличии от типа char, string является классом. Теперь чтоб создать строку достаточно написать: string s; Для записи в строку можно использовать оператор = s="Hello"; Пример работы с классом string: string name; cout<<"Enter your name"<<endl; cin>>name; cout<<"Hi "<<s<<"!"<<endl; Существует множество функций для работы со строками. • s.append(str) - добавляет в конец строки строку str. Можно писать как s.append(переменная), так и s.append("строка"); • s.assign(str) - присваивает строке s значение строки str. Аналогично записи s=str; • int i=s.begin() - записывает в i индекс первого элемента строки • int i=s.end() - аналогично, но последнего • s.clear() - как следует из названия, отчищает строку. Т.е. удаляет все элементы в ней • s.compare(str) -сравнивает строку s со строкой str и возвращает 0 в случае совпадение (на самом деле сравнивает коды символов и возвращает из разность) • s.copy(куда, сколько, начиная с какого) - копирует из строки s в куда

(там может быть как строка типа стринг, так и строка типа char). Последние 2 параметра не обязательные (можно использовать функцию с 1,2 или 3 параметрами) • bool b=s.empty() - если строка пуста, возвращает true, иначе false • s.erase(откуда, сколько) удаляет n элементов с заданной позиции • s.find(str,позиция) - ищет строку str начиная с заданной позиции • s.insert(позиция,str, начиная, beg, count) - вставляет в строку s начиная с заданной позиции часть строки str начиная с позиции beg и вставляя count символов • int len=s.length() - записывает в len длинну строки • s.push_back(symbol) - добавляет в конец строки символ • s.replace(index, n,str) - берет n первых символов из str и заменяет символы строки s на них, начиная с позиции index • str=s.substr(n,m) - возвращает m символов начиная с позиции n • s.swap(str) меняет содержимое s и str местами. • s.size() - возвращает число элементов в строке. Вот собственно большинство необходимых функция для работы со строками в c++.

Лекция 13. Графика. Консольная графика. GUI. (см. слайды П_1_13_Графика)

Графика в консоли Казалось бы, текстовый режим системной консоли и работа с графикой суть вещи несовместные. И действительно, консоль — не самое подходящее место для редактирования полноцветных фотореалистических изображений или сложной векторной графики. Один из вариантов «украшения» консольного приложения — использование библиотеки ncurses. Ncurses — библиотека написанная на языке Си и предназначенная для управления вводом/выводом натерминал, в том числе — задавать экранные координаты (в знакоместах) и цвет выводимых символов. Предоставляет программисту уровень абстракции, позволяющий не беспокоиться об аппаратных различиях терминалов и писать переносимый код. Основными концепциями пользовательского интерфейса программы, использующей ncurses, являются экран (screen), окно (window) и под-окно (sub-window). Экраном называется все пространство, на котором ncurses может выводить данные. С точки зрения ncurses, экран – это матрица ячеек, в которые можно выводить символы. Если монитор работает в текстовом режиме, экран ncurses совпадает с экраном монитора. Если терминал эмулируется графической программой, экраном является рабочая область окна этой программы. Окном ncurses называется прямоугольная часть экрана, для которой определены особые параметры вывода. В частности, размеры окна влияют на перенос и прокрутку строк, выводимых в этом окне. В каком-то смысле окно можно назвать «экраном в экране». На уровне интерфейса программирования окна представлены структурами данных, по этой причине мы будем часто говорить об окне как о структуре. В процессе инициализации ncurses автоматически создается окно stdscr, размеры которого совпадают с размерами экрана. Кроме структуры stdscr по умолчанию создается еще одна структура – curscr. Операции вывода данных ncurses модифицируют содержимое структуры stdscr, однако на экране всегда отображается содержимое окна curscr. Иначе говоря, данные, которые выводит ваша программа в окно stdscr (или в другое окно), не отображаются на экране монитора автоматически. Для того чтобы сделать результаты вывода видимыми, вы должны вызывать специальные функции обновления экрана (refresh() или wrefresh()). Эти функции сравнивают содержимое окон stdscr и curscr и на основе различий между ними вносят изменения в структуру curscr, а затем обновляют экран. Благодаря наличию окна curscr, ncurses-программе не требуется «помнить» весь свой предыдущий вывод и перерисовывать его всякий раз, когда в этом возникает необходимость. Этим программы ncurses отличаются от графических программ. В старину, когда терминалы связывались с компьютерами через модемы, использование двух окон давало дополнительное преимущество в скорости обмена данными, ведь программе нужно было передавать на терминал не копию экрана целиком, а только разницу между содержимым окон curscr и stdscr. Хотя ваша программа может пользоваться для вывода данных исключительно окном stdscr, ваша задача по проектированию интерфейса существенно упростится, если вы будете создавать собственные окна, расположенные «внутри» stdscr. Программа, использующая ncurses, может работать с несколькими окнами одновременно, выполняя вывод в каждое из них. Кроме окон (windows) программы ncurses могут создавать под-окна

(subwindows), поведение которых несколько отличается от поведения стандартных окон. Важнейшей особенностью ncurses является возможность указать произвольную позицию курсора для вывода (и ввода) данных. Позиция курсора отсчитывается от левого верхнего угла текущего окна. Ячейка в верхнем левом углу имеет координаты (0, 0). При работе с функциями ncurses важно помнить, что первой координатой является номер строки (что соответствует y, в терминах графического программирования), а второй координатой – номер столбца (что соответствует x в графическом режиме).
В случае ошибки функции ncurses обычно возвращают константу ERR. Если функция не должна возвращать какое-то информативное значение (как, например, функция getch()), в случае успешного выполнения она возвращает значение OK. Для win32-приложений существует аналог библиотеки ncurses – PDCurses. Листинг программы “Hello, world” с использованием PDCurses: #include <curses.h> #include <string.h> #define WHEIGHT 5 #define WWIDTH 40 int main () { WINDOW *my_win; char *str="Hello, world!"; initscr(); start_color(); if (has_colors()) { init_pair(1, COLOR_CYAN, COLOR_RED); } my_win=newwin(WHEIGHT, WWIDTH, (LINES-WHEIGHT)/2, (COLS-WWIDTH)/2); box(my_win, '*', '*'); wbkgd(my_win, COLOR_PAIR(1)|A_BOLD|A_BLINK); mvwaddstr(my_win, WHEIGHT/2, (WWIDTH-strlen(str))/2, str); wrefresh(my_win); endwin(); return 0; } Существует также библиотека conio, отвечающая за консольный ввод-вывод (на данный момент, устаревшая). conio.h — заголовочный файл, используемый в старых компиляторах, работающих в операционных системах MS-DOS, для создания текстового интерфейса пользователя. Пример с ипользованием этой библиотеки: #include <conio.h> #include <iostream> #include <windows.h> using namespace std; int main(){ SetConsoleTitle("boolean"); WORD ATTR; COORD POS; HANDLE CH=GetStdHandle(STD_OUTPUT_HANDLE); for(int items=0;items<100;items++){ ATTR=FOREGROUND_BLUE*(rand()%2) | FOREGROUND_GREEN | FOREGROUND_INTENSITY*(rand()%2) | BACKGROUND_BLUE*(rand()%2); POS.X=rand()%70; POS.Y=rand()%20; SetConsoleTextAttribute(CH,ATTR); SetConsoleCursorPosition(CH,POS); cout<<char(1+rand()%6)<<endl; } POS.X=0; POS.Y=21; SetConsoleCursorPosition(CH,POS); getch();
return 0; }

GUI Графический интерфейс пользователя (англ. Graphical user interface, GUI) — разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений. В отличие от интерфейса командной строки, в GUI пользователь имеет произвольный доступ (с помощью устройств ввода — клавиатуры, мыши, джойстика и т. п.) ко всем видимым экранным объектам (элементам интерфейса) и осуществляет непосредственное манипулирование ими. Чаще всего элементы интерфейса в GUI реализованы на основе метафор и отображают их назначение и свойства, что облегчает понимание и освоение программ неподготовленными пользователями. Одним из требований к хорошему графическому интерфейсу программной системы является концепция «делай то, что я имею в виду» или DWIM (англ. Do What I Mean). DWIM требует, чтобы система работала предсказуемо, чтобы пользователь заранее интуитивно понимал, какое действие выполнит программа после получения его команды. Существует огромное количество библиотек, фреймворков, виджетов, предназначенных для создания элементов графического интерфейса к программам на C++: • QT • WTL • ATL (Adam&Eve) • eGUI++ • Juce • Ultimate++ • Smartwin++ • FOX • SolidWidgets и многие другие. В целом, изучение конкретных графических библиотек выходит за рамки данного курса, а выбор в пользу той или иной технологии создания интерфейсов остается за студентами.
Лекция 14. API-функции. Использование системных методов. (см. слайды П_1_14_API)
Windows API Windows API — общее наименование целого набора базовых функций

интерфейсов программирования приложений операционных систем семейств Microsoft Windows корпорации «Майкрософт» и совместимой с ними свободной бесплатной операционной системы ReactOS. Windows API был изначально спроектирован для использования в программах, написанных на языке Си/C++. Работа через Windows API — это наиболее близкий к системе способ взаимодействия с ней из прикладных программ. Более низкий уровень доступа, необходимый только для драйверов устройств, в текущих версиях Windows предоставляется черезWindows Driver Model. Примеры #include <windows.h> int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow) { MessageBox(NULL, "Сказала собака баскервилей.", "Ну, здравствуй, Герасим.", MB_OK| MB_ICONEXCLAMATION); return 0; } Начнем с параметров главной функции. Кстати, это САМАЯ главная функция. Функция WndProc (о которой позже) является главной оконной процедурой. Итак, параметры… HINSTANCE вообще интересный тип. Это своего рода дескриптор объекта (HANDLE). В первом случае (hInstance) он указывает на тот исполняемый модуль, на основе которого запущен процесс данного кода. Проще говоря, указывает на файл “NoWindow.exe”. Второй параметр (hPrevInstance) – пережиток Win16. В 32-разрядных приложениях не используется. Раньше обозначал предыдущий запущенный экземпляр данного приложения. Третий (lpszCmdParam) – строка параметров, которые были посланы программе при запуске. Если их не было, то строка пуста. Четвертый (nCmdShow) – число, обозначающее вид окна при запуске. Например, свернутое, развернутое, нормальное. За одно, разберем параметры вызова MessageBox. Первый – дескриптор окна, к которому принадлежит сообщение. Это нужно для того, что бы Windows знала, какое окно его вызвало и, соответственно, сделать его неактивным, пока вы не ответите в нем нажатием какой-нибудь кнопки. Второй и третий – строки. Соответственно, текста сообщения и текста заголовка сообщения. Четвертый – самый интересный. В нем указывается стиль сообщения. Они находятся в файле <WinUser.h>

#include <windows.h> #include <windowsx.h> #define SimWnd_DefProc DefWindowProc LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam) ; BOOL Register(HINSTANCE hInstance); HWND Create(HINSTANCE hInstance, int nCmdShow); static char szAppName[] = "SimWnd"; static HWND hMainWindow;

void SimWnd_OnDestroy(HWND hwnd); int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow) { MSG Msg; if (!Register(hInstance)) return FALSE; if (!Create(hInstance, nCmdShow)) return FALSE; while (GetMessage(&Msg, NULL, 0, 0)) { TranslateMessage(&Msg); DispatchMessage(&Msg); } return Msg.wParam; } BOOL Register(HINSTANCE hInstance) { WNDCLASS WndClass; WndClass.style = CS_HREDRAW | CS_VREDRAW; WndClass.lpfnWndProc = WndProc; WndClass.cbClsExtra = 0; WndClass.cbWndExtra = 0; WndClass.hInstance = hInstance; WndClass.hIcon = LoadIcon(NULL, IDI_APPLICATION); WndClass.hCursor = LoadCursor(NULL, IDC_ARROW); WndClass.hbrBackground = (HBRUSH)(COLOR_BTNFACE+1); WndClass.lpszMenuName = NULL; WndClass.lpszClassName = szAppName; return (RegisterClass(&WndClass) != 0); } HWND Create(HINSTANCE hInstance, int nCmdShow) { HWND hwnd = CreateWindow(szAppName, szAppName, WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL); if (hwnd == NULL) return FALSE; ShowWindow(hwnd, nCmdShow); UpdateWindow(hwnd); return hwnd; } LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam) { switch (Message) { HANDLE_MSG(hwnd, WM_DESTROY, SimWnd_OnDestroy); default: return SimWnd_DefProc(hwnd, Message, wParam, lParam); } }

void SimWnd_OnDestroy(HWND hwnd) { PostQuitMessage(0); } Три шага создания окна: 1) Регистрация. (Register()) 2) Создание. (Create()) 3) Обработка сообщений. (WndProc()) Когда

мы создаем оконное приложение, мы как бы делаем подкласс объекта «окно» благодаря этим трем шагам. При создании кнопки (класс “button”) , регистрацию и обработку сообщений выполняет система, а создание – программист. Программист также может создать свой класс кнопки и сделать ей любые свойства такие, как подсветка или градиент. При регистрации, мы сообщаем системе, в основном, название нового класса и процедуру обработки его сообщений, ну и некоторые элементы внешнего вида и другие параметры. Создание окна – тут все просто. Вызов функции CreateWindow(). А при создании нового окна, после этой вызываются ShowWindow() и UpdateWindow(). Они призваны соответственно показать и обновить окно. Самое сложное и обычно наиболее емкое по коды – это обработка сообщений. Есть такая штука – DefWindowProc – стандартная оконная процедура. Все наше окно полностью работает за счет этой процедуры. Сообщения. Все функционирование в Windows основано на сообщениях. Даже то, что вы передвигаете мышь. Точнее её перерисовка. А как их получает оконная процедура? Все начинается тут: while (GetMessage(&Msg, NULL, 0, 0)) { TranslateMessage(&Msg); DispatchMessage(&Msg); }
А продолжается тут: LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam) { switch (Message) { HANDLE_MSG(hwnd, WM_DESTROY, SimWnd_OnDestroy); default: return SimWnd_DefProc(hwnd, Message, wParam, lParam); } } Функция GetMessage получает сообщение от системы и пересылает его оконной процедуре. Там программист обрабатывает его так, как хочет. Если он этого не делает или ему этого не нужно, то оно обрабатывается стандартной процедурой. Сообщений очень много и приводить их здесь нецелесообразно – займет пару страниц. Скажу только то, что когда я не знал, как называется сообщение, то открывал файл <WinUser.h> искал WM_NULL и смотрел далее сообщения, а по названию определял назначение, далее по справке и, в конце-концов, можно найти. Рассмотрим параметры в функции WndProc: 1) Дескриптор окна, к которому относится сообщение. 2) Само сообщение. 3) Параметр сообщения.
4) Параметр сообщения. Использование системных методов Операционная система windows является объектно-ориентированной. Все элементы управления (объекты) являются окнами в том или ином виде. Каждый такой элемент имеет свои параметры состояния, входные и выходные сообщения. Традиционно при написании программ с использованием чистого winapi применяются методы структурного и объектноориентированного программирования.
Лекция 15. Обработка ошибок и исключений (см. слайды П_1_15_Исключения)
Обработка исключений 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. Давайте же разберем основы обработки исключений в C++. Чтобы комфортно работать с исключениями в C++ вам нужно знать лишь три ключевых слова: • try (пытаться) - начало блока исключений; • catch (поймать) - начало блока, "ловящего" исключение; • throw (бросить) - ключевое слово, "создающее" ("возбуждающее") исключение. А теперь пример, демонстрирующий, как применить то, что вы узнали: void func() { try { throw 1; } catch(int a) { cout << "Caught exception number: " << a << endl; return; } cout << "No exception detected!" << endl; return; } Если выполнить этот фрагмент кода, то мы получим следующий результат: Caught exception number: 1 Теперь закоментируйте строку throw 1; и функция выдаст такой результат: No exception detected! Как видите все очень просто, но если это применить с умом, такой подход покажется вам очень мощным средством обработки ошибок.

Catch может "ловить" любой тип данных, так же как и throw может "кинуть" данные любого типа. Т.е. throw AnyClass(); будет правильно работать, так же как и catch (AnyClass &d) {};. Как уже было сказано, catch может "ловить" данные любого типа, но вовсе не обязательно при это указывать переменную. Т.е. прекрасно будет работать что-нибудь типа этого: catch(dumbclass) { } так же, как и catch(dumbclass&) { } Так же можно "поймать" и все исключения: catch(...) { } Троеточие в этом случае показывает, что будут пойманы все исключения. При таком подходе нельзя указать имя переменной. В случае, если "кидаются" данные нестандартного типа (экземпляры определенных вами классов, структур и т.д.), лучше "ловить" их по ссылке,
иначе вся "кидаемая" переменная будет скопирована в стек вместо того, чтобы просто передать указатель на нее. Если кидаются данные нескольких типов и вы хотите поймать конкретную переменную (вернее, переменную конкретного типа), то можно использовать несколько блоков catch, ловящих "свой" тип данных: try { throw 1; // throw 'a'; } catch (long b) { cout << "пойман тип long: " << b << endl; } catch (char b) { cout << "пойман тип char: " << b << endl; } «Создание» исключений Когда возбуждается исключительная ситуация, программа просматривает стек функций до тех пор, пока не находит соответствующий catch. Если оператор catch не найден, STL будет обрабатывать исключение в стандартном обработчике, который делает все менее изящно, чем могли бы сделать вы, показывая какие-то непонятные (для конечного пользователя) сообщения и обычно аварийно завершая программу. Однако более важным моментом является то, что пока просматривается стек функций, вызываются деструкторы всех локальных классов, так что вам не нужно заботиться об освобождении памяти и т.п. Перегрузка глобальных операторов new/delete Если перегрузить стандартные new и delete, то открываются широкие возможности по отслеживанию ошибок (причем ошибок часто критических) с помощью исключений. Например: char *a; try { a = new char[10]; } catch (...){ // а не создан - обработать ошибку распределения памяти, // выйти из программы и т.п. } // а успешно создан, продолжаем выполнение Операторы throw без параметров Итак, мы увидели, как новый метод обработки ошибок удобен и прост. Блок try-catch может содержать вложенные блоки try-catch и если не будет определено соответствующего оператора catch на текущем уровен вложения, исключение будет поймано на более высоком уровне. Единственная вещь, о которой вы должны помнить, - это то, что операторы, следующие за throw, никогда не выполнятся. try { throw; // ни один оператор, следующий далее (до закрывающей скобки) // выполнен не будет } catch(...) { cout << "Исключение!" << endl;
} Такой метод может применяться в случаях, когда не нужно передавать никаких данных в блок catch. Перехват исключений #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 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 catchпредложение, 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. 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. А если catch-обработчики не содержат инструкции return, с какого места будет продолжено выполнение программы? 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 catchобработчик 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. 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 объектеисключении 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. Исключения и вопросы проектирования С обработкой исключений в программах C++ связано несколько вопросов. Хотя поддержка такой обработки встроена в язык, не стоит использовать ее везде. Обычно она применяется для обмена информацией об ошибках между независимо разработанными частями программы. Например, автор некоторой библиотеки может с помощью исключений сообщать пользователям об ошибках. Если библиотечная функция обнаруживает аномальную ситуацию, которую не способна обработать самостоятельно, она может возбудить исключение для уведомления вызывающей программы. В нашем примере в библиотеке определен класс iStack и его функции-члены. Разумно предположить, что программист, кодировавший main(), где используется эта библиотека, не разрабатывал ее. Функции-члены класса iStack могут обнаружить, что операция pop() вызвана, когда стек пуст, или что операция push() вызвана,

когда стек полон; однако разработчик библиотеки ничего не знал о программе, пользующейся его функциями, так что не мог разрешить проблему локально. Не сумев обработать ошибку внутри функцийчленов, мы решили возбуждать исключения, чтобы известить вызывающую программу. Хотя C++ поддерживает исключения, следует применять и другие методы обработки ошибок (например, возврат кода ошибки) – там, где это более уместно. Однозначного ответа на вопрос: "Когда ошибку следует трактовать как исключение?" не существует. Ответственность за решение о том, что считать исключительной ситуацией, возлагается на разработчика. Исключения – это часть интерфейса библиотеки, и решение о том, какие исключения она возбуждает, – важный аспект ее дизайна. Если библиотека предназначена для использования в программах, которые не должны аварийно завершаться ни при каких обстоятельствах, то она обязана разбираться с аномалиями сама либо извещать о них вызывающую программу, передавая ей управление. Решение о том, какие ошибки следует обрабатывать как исключения, – трудная часть работы по проектированию библиотеки. В нашем примере с классом iStack вопрос, должна ли функция push() возбуждать исключение, если стек полон, является спорным. Альтернативная и, по мнению многих, лучшая реализация push() – локальное решение проблемы: увеличение размера стека при его заполнении. В конце концов, единственное ограничение – это объем доступной программе памяти. Наше решение о возбуждении исключения при попытке поместить значение в полный стек, по-видимому, непродуманно. Можно переделать функцию-член push(), чтобы она в такой ситуации наращивала стек: void iStack::push(int value) { // если стек полон, увеличить размер вектора if (full())

_stack.resize(2 * _stack.size()); _stack[_top++] = value; } Аналогично следует ли функции pop() возбуждать исключение при попытке извлечь значение из пустого стека? Интересно отметить, что класс stack из стандартной библиотеки C++ не возбуждает исключения в такой ситуации. Вместо этого постулируется, что поведение программы при попытке выполнения подобной операции не определено. Разрешить программе продолжать работу при обнаружении некорректного состояния признали возможным. Мы уже упоминали, что в разных библиотеках определены разные исключения. Не существует пригодного для всех случаев ответа на вопрос, что такое исключение. Не все программы должны беспокоиться по поводу исключений, возбуждаемых библиотечными функциями. Хотя есть системы, для которых простой недопустим и которые, следовательно, должны обрабатывать все исключительные ситуации, не к каждой программе предъявляются такие требования. Обработка исключений предназначена в первую очередь для реализации отказоустойчивых систем. В этом случае решение о том, должна ли программа обрабатывать все исключения, возбуждаемые библиотеками, или может закончить выполнение аварийно, – это трудная часть процесса проектирования. Еще один аспект проектирования программ заключается в том, что обработка исключений обычно структурирована. Как правило, программа строится из компонентов, и каждый компонент решает сам, какие исключения обрабатывать локально, а какие передавать на верхние уровни. Что мы понимаем под компонентом? Например, система анализа текстовых запросов может быть разбита на три компонента, или слоя. Первый слой – это стандартная библиотека C++, которая обеспечивает базовые операции над строками, отображениями и т.д. Второй слой – это сама система анализа текстовых запросов, где определены такие функции, как string_caps() и suffix_text(), манипулирующие текстами и использующие стандартную библиотеку как основу. Третий слой – это программа, которая применяет нашу систему. Каждый компонент строится независимо и должен принимать решения о том, какие исключительные ситуации обрабатывать локально, а какие передавать на более высокий уровень. Не все функции должны уметь обрабатывать исключения. Обычно try-блоки и ассоциированные с ними catch-обработчики применяются в функциях, являющихся точками входа в компонент. Catch-обработчики проектируются так, чтобы перехватывать те исключения, которые не должны попасть на верхние уровни программы. Для

этого также используются спецификации исключений.
Лекция 16. Оптимизация кода. Интеграция с другими языками программирования. (см. слайды П_1_16_Оптимизация_кода)
Оптимизация кода Некоторые считают, что времена оптимизации на уровне кода прошли навсегда, однако это не так. Сейчас существует множество платформ в которых нет таких могущественных компиляторов как в Microsoft Visual Studio. Например шейдерные языки (hlsl, glsl) или код для CUDA, PlayStation3, SPU или мобильные платформы. В зависимости от организации кода, может в десятки раз отличаться его эффективность иногда из-за неэффективности компилятора, на чаще из-за доступа к памяти. Программируя для разных платформ изучите возможности вашего компилятора и особенности архитектуры процессора (если вы пишите для конкретной консоли). Проведите тесты производительности разных вариантов оптимизации. Часто сложно предполагать, какие именно способы будут наиболее эффективными. А эта статья подскажет разные приемы, которые могут вам помочь. Однако не следует оптимизировать слепо, без предварительного анализа и профилирования. Помните, что преждевременная оптимизация — это зло. 1. Используйте векторизацию данных и векторные команды их обработки (например SSE в CPU или упаковывайте данные если используете шейдеры или CUDA). Это позволит использовать SIMD (Single Instruction, Multiple Data) архитектуру, что значительно повысит скорость вычислений. Если вы решите использовать этот метод, то не забывайте про выравнивание данных в памяти. 2. Эффективнее складывать и умножать в перемешку, чем сначала все сложить, а потом все умножить. Это происходит от того, что сложение и умножение выполняются разными модулями процессора и могут выполняться одновременно. int [float, double, single, unsigned] a,b,c,k,m,n,t,f1,f2,f3,g1,g2,g3; a = b + c; k = m + n; t = a + k; f1 = f2 * f3; g1 = g2 * g3; Менее эффективно чем: a = b + c; f1 = f2 * f3; k = m + n; g1 = g2 * g3; t = a + k; 3. Нет разницы по скорости в работе с float и double при сложении и умножении. Они выполняются за равное число тактов процессора и используют одни и те же регистры. При делении и извлечении корня, float работает быстрее. Однако если вы будете использовать большие объемы данных, то за счет кеша быстрее будет работать тот тип, который занимает меньше памяти (т.е. float), поэтому в общем случае именно его предпочтительно использовать. Выбирать double имеет смысл когда нужна большая точность. 4. Допустим, есть const float a = 100.0f; float some1 = some3 * 1.0f / a; float some2 = some4 * 1.0f / a; более эффективно написать не: const float a_inv = 1.0f / a; some1 = some3 * a_inv;
some2 = some4 * a_inv; а так: some1 = some3 * (1.0f / a); some2 = some4 * (1.0f / a); Почему это будет эффективнее? Операторы с равным приоритетом выполняются последовательно. Это значит, что будет выполнено сначала умножение, а затем деление. Если же обрамить операцию деления в скобки, то ее выполнит компилятор, а в реальном времени будет выполняться только операция умножения. Что качается отличий варианта 3 от варианта 2, то в 3-ем варианте не создается дополнительной переменной, нет нужны глядя на код думать о том, что это за переменная. А эффективность 2-го и 3-го варианты будет одинаковой. 5. На больших объемах данных и вычислений на них float выгоднее чем double (из за cache miss'ов, см. пункт 3). 6. a, b - любое выражение Func1, Func2 - функции, которые вызовутся для вычисления условия if(a && b) - нужно первым ставить менее вероятное условие if(a || b) - нужно первым ставить более вероятное условие if(Func1() && Func2()) - нужно ставить более быстрый оператор первым 7. void Func(int* a) { int b = 10; Следующие строки одинаковые по эффективности (по времени выполнения): b = 100; *a = 100; } Это происходит по тому, что доступ к стековой переменной осуществляется по указателю на стек. Тоже идет разыменование указателя. 8. Если есть большой массив структур, то нужно делать размер его элементов равным степени двойки. Тогда проход по такому массиву будет значительно быстрее (в 4 -6 раз), за счет выравнивания указателя на каждую структуру в массиве. 9. int a[1000]; for(int i =0; i<1000; ++i) a[i] = 50; Значительно эффективнее будет: int* p = a; for(int

i =0; i<1000; ++i, ++p) *p = 50; 10. SomeClass* p; - указатель на массив элементов x = *(p++); - значительно эффективнее x = *(++p); По той же причине что и пункт 1. В первом случае будет осуществляться разыменование указателя и его инкремент параллельно, а во втором — последовательно. 11. Количество колонок двумерного массива желательно должно быть равно степени двойки. Это увеличит скорость работы с массивом. Это выравняет указатели на первые элементы каждой строки, что ускорит доступ к элементам. И так далее. Существует большое количество вариантов оптимизации кода за счет эффективного использования памяти. Компромиссы (tradeoff) Оптимизация в основном фокусируется на одиночном или повторном времени

выполнения, использовании памяти, дискового пространства, пропускной способности или некотором другом ресурсе. Это обычно требует компромиссов — один параметр оптимизируется за счёт других. Например, увеличение размера программного кеша чеголибо улучшает производительность времени выполнения, но также увеличивает потребление памяти. Другие распространённые компромиссы включают прозрачность кода и его выразительность, почти всегда ценой деоптимизации. Сложные специализированные алгоритмы требуют больше усилий по отладке и увеличивают вероятность ошибок. Узкие места Для оптимизации требуется найти узкое место (hotspot), иногда называемое бутылочным горлышком (bottleneck): критическую часть кода, которая является основным потребителем необходимого ресурса. Улучшение примерно 20 % кода иногда влечёт за собой изменение 80 % результатов. Для поиска узких мест используются специальные программы — профайлеры. Утечка ресурсов (памяти, дескрипторов и т. д.) также может привести к падению скорости выполнения программы, для их нахождения также применяются специальные программы. Приёмы оптимизации программ по затратам процессорного времени Оптимизация по затратам процессорного времени особенно важна для расчетных программ, в которых большой удельный вес имеют математические вычисления. Здесь перечислены некоторые приемы оптимизации, которые может использовать программист при написании исходного текста программы. 1. Инициализация объектов данных Во многих программах какую-то часть объектов данных необходимо инициализировать, то есть присвоить им начальные значения. Такое присваивание выполняется либо в самом начале программы, либо, например, в конце цикла. Правильная инициализация объектов позволяет сэкономить драгоценное процессорное время. Так, например, если речь идет об инициализации массивов, использование цикла, скорее всего, будет менее эффективным, чем объявление этого массива прямым присвоением. 2. Программирование арифметических операций В том случае, когда значительная часть времени работы программы отводится арифметическим вычислениям, немалые резервы повышения скорости работы программы таятся в правильном программировании арифметических (и логических) выражений. Важно, что различные арифметические операции значительно различаются по быстродействию. В большинстве архитектур, самыми быстрыми являются операции сложения и вычитания. Более медленным является умножение, затем идёт деление. Относительно много времени тратится на обращение к подпрограммам (передача параметров через стек, сохранение регистров и адреса возврата, вызов конструкторов копирования). Если подпрограмма содержит малое число действий, она может быть реализована подставляемой (англ. inline) — все её операторы копируются в каждое новое место вызова (существует ряд ограничений на inline-подстановки: например, подпрограмма не должна быть рекурсивной). Это ликвидирует накладные расходы на обращение к подпрограмме, однако ведет к увеличению размера исполняемого файла. Само по себе увеличение размера исполняемого файла не является существенным, однако в некоторых случаях исполняемый код может выйти за пределы кэша команд, что повлечет значительное падение скорости исполнения программы. Поэтому современные оптимизирующие компиляторы обычно имеют настройки оптимизации по размеру кода и по скорости выполнения. 3. Циклы Различается и время выполнения циклов разного типа. Время

выполнения цикла со счетчиком и цикла с постусловием при всех прочих равных условиях совпадает, цикл с предусловием выполняется несколько дольше (примерно на 20-30 %). При использовании вложенных циклов следует иметь в виду, что затраты процессорного времени на обработку такой конструкции могут зависеть от порядка
следования вложенных циклов. Можно сделать вывод, что при программировании вложенных циклов по возможности следует делать цикл с наибольшим числом повторений самым внутренним, а цикл с наименьшим числом повторений — самым внешним. Если в циклах содержатся обращения к памяти (обычно при обработке массивов), для максимально эффективного использования кэша и механизма аппаратной предвыборки данных из памяти порядок обхода адресов памяти должен быть по возможности последовательным. Классическим примером подобной оптимизации является смена порядка следования вложенных циклов при выполнении умножения матриц. 4. Инвариантные фрагменты кода Оптимизация инвариантных фрагментов кода тесно связана с проблемой оптимального программирования циклов. Внутри цикла могут встречаться выражения, фрагменты которых никак не зависят от управляющей переменной цикла. Их называют инвариантными фрагментами кода. Современные компиляторы часто определяют наличие таких фрагментов и выполняют их автоматическую оптимизацию. Такое возможно не всегда, и иногда производительность программы зависит целиком от того, как запрограммирован цикл. Ассемблерные вставки Существует два способа вставить ассемблерный код в программу на c/c++ - встроенный ассемблер (inline) и подключение .asm-файла. Inline-подключение ассемблерного кода зависит от компилятора и выглядит либо так: asm("ассемблерная строка"); или __asm__("ассемблерная строка"); //безопасный метод либо: _asm { // код на Assembler } Пример: int main(void) { int foo = 10, bar = 15; __asm__ __volatile__("addl %%ebx,%%eax" :"=a"(foo) :"a"(foo), "b"(bar)); printf("foo+bar=%d\n", foo); return 0; } Совместное применение: C++ и Java Язык Java был создан на основе языка C++, у которого были позаимствованы синтаксис и базовая семантика. В Java используются практически идентичные соглашения для объявления переменных, передачи параметров, операторов и для управления потоком выполнением программы. В Java были добавлены все хорошие черты C++, но исключены некоторые недостатки последнего. Например, оператор goto, глобальные переменные и ненадежное приведение типов. Более того, Java полностью объектно-ориентированный язык. Однако все это привело к уменьшению быстродействия программ, невозможности напрямую использовать системные ресурсы, а также к некоторым другим проблемам. Сегодня возможностей языка Java и его стандартных библиотек классов вполне достаточно для написания различных полнофункциональных приложений. В Java JDK входит множество библиотек классов, в которых есть все, начиная от классов ввода/вывода и кончая классами
для распределенного программирования. Но бывают случаи, когда программисту необходимы возможности, не поддерживаемые Java, но имеющиеся в C++. Например, использовать ресурсы конкретной операционной системы, взаимодействовать с различными аппаратными устройствами, пользоваться уже существующим, написанными не на Java кодом, или создавать программы реального времени. Для этого в Java есть интерфейс, который используется для вызова функций, написанных на других языках. Сегодня такой интерфейс существует только для языка C++. Механизм языка Java используемый для вызова кода, написанного на других языках, называется JNI (Java Native Interface).
Лекция 17. Языки программирования. История и перспективы развития. (см. слайды П_1_17_ЯП)
История развития языков программирования С чего все начиналось В далекие-далекие времена, когда еще не было современных языков программирования, люди, а точнее программисты тех времен, имели дело непосредственно с машинным кодом. Естественно, такие люди очень ценились, и к ним предъявлялись огромные требования, такие, как

превосходное знание машинного кода, всей архитектуры тех компьютеров (напомню, что основным носителем информации был не винт, как сейчас, а обыкновенные перфокарты и перфоленты). Но все-таки программы того времени были достаточно простые из-за ряда причин, таких, как сложность машинного кода, его отладки и, конечно, самой примитивности вычислительных машин. И первым шагом у компьютерной цивилизации был переход к Ассемблеру. Assembler Может показаться, что это событие не было из ряда вон выходящим: мол, как были машинные коды, так и остались. Однако это не так. Да, программирование в машинных кодах осталось, но она стало символическим кодированием машинных команд. Это был огромный шаг вперед, теперь программисту вовсе не нужно было разбираться в программировании команд на аппаратном уровне. В программировании на Ассемблере стало возможным использование меток, что облегчало в значительной мере отладку программ. Появилась возможность разработки целой серии вычислительных машин с одинаковой или сходной системой команд, что можно назвать первым подобием переносимости кода. Ассемблер нес в себе еще одну революционность для программирования: появилось два представления программ на нем: в машинных кодах и в чистом, откомпилированном виде. Но, по мере развития Ассемблера, его постоянного усложнения, дизассемблирование, т.е. перевод из одного представления программы в другой, становилось все более трудным делом. Надо отметить, что в итоге автоматическое дизассемблирование стало невозможным - для этой цели стали появляться специальные программы-дизассемблеры. Fortran Первый язык программирования высокого уровня под названием Фортран (Fortran) был разработан в 1954 году Джоном Бэкусом, работником корпорации IBM в Нью-Йорке. Данное событие, как ты понимаешь, было очень значимым: впервые в истории человечества программист мог отвлечься от разборок со сложными машинными кодами и вздохнуть с облегчением =). Самым главным и принципиальным отличием Фортрана от Ассемблера была концепция подпрограмм. Хотя в наше время это и не в новинку, тогда это было действительно новым. Для разработчиков достигнуть этого стоило небывалых усилий: скажу лишь, что сама работа над компилятором продолжалась более двух лет! В апреле 1957 года компилятор был полностью готов и мог быть использован на машинах IBM-704. Несмотря на это, большинство программистов встретили Фортран недовольно. Они считали гораздо более эффективным и надежным написать небольшой код вручную, чем получить его как часть компиляции. Но время идет, требования к жизни становятся жестче: не обошло это и программирование. Вскоре стало понятно, что компилятор для программиста не враг, а друг, особенно это становится заметно при реализации больших проектов. Преимуществом Фортрана было и то, что он был достаточно легким для обучения. Вскоре он стал более или менее признан и использовался на шести различных моделях компьютеров фирмы IBM. Но, несмотря на все это, новый язык не был лишен и недостатков, что, в конечном счете, привело к появлению Фортрана II. Одним из основных преимуществ новой версии над старой была способность присоединять программы на ассемблере. Затем создавались еще несколько версий этого языка вплоть до Фортрана версии IV. Этот язык программирования использовался (и используется по сей день) для различных научных расчетов и вычислений. Роль Фортрана была огромна, и одна из них дала понять всем ведущим программистам, что
будущее за языками высокого уровня. Cobol Вскоре после Фортрана, а именно в 1960 году, был создан язык Кобол (Cobol). Он изначально разрабатывался для создания коммерческих приложений. Даже расшифровывается как Common Business Oriented Language - универсальный язык, предназначенный для бизнеса. Все началось в далеком 1959 году, когда прошла конференция по языкам программирования - CODASYL. Причиной ее созыва была причина создания единого языка для деловых приложений. Что и было реализовано в Коболе. Язык сразу признали достаточно удобным для любой сферы бизнеса, был очень удачен для людей, не особо искушенных в программировании, и, что немаловажно, был машинно-независим. Его популярность заметно увеличилась после выхода в 1962 году улучшенной

второй версии языка. Вследствие этого, на нем были написаны многие тысячи коммерческих прикладных программ. Одной из главных особенностей этого языка стала возможность эффективной работы с большими массивами данных, что характерно именно для коммерческих и деловых приложений. В целом, стоит отметить, что, несмотря на многие сходства с Фортраном, этот язык программирования был принят гораздо легче по сравнению с тем же Фортраном. PL/1 Все той же компанией IBM в 1964 году был разработан новый язык программирования - NPL (New Programming Language). Но, дабы избежать путаницы с той же аббревиатурой у Национальной Физической Лаборатории в Англии (National Physical Laboratory), язык был переименован в PL/1. Целью его разработки ставилось замещение новым языком программирования Фортрана и Кобола в большинстве приложений. До него исключительно все языки программирования, взять тот же Кобол, разрабатывались для какой-либо отдельной области, например, науки. PL/1 же, в отличие от них, был призван использоваться в более широких областях применения. Это было его отличием; отличием вроде бы достаточным, чтобы все сказали о нем: "PL/1 - номер один в мире!", но этого не произошло. Вроде бы обладая огромным богатством синтаксических конструкций, универсальностью применения, он мог бы претендовать на это место. Но дело в том, что, несмотря на богатство синтаксиса языка, он был весьма сложным, опять-таки в отличие от предыдущих языков программирования. По этой причине разработка компилятора под него была крайне сложным делом. Поэтому язык всемирно и не распространился, а использовался лишь внутри корпорации IBM. Basic В 1963 году Джон Кемени и Томас Курц, сотрудники математического факультета Дармутского колледжа, написали простой и легкий в изучении язык, который назвали Бейсик (Basic Beginner's All-purpose Symbolic Instruction Code - универсальный код символических инструкций для начинающих). Язык задумывался, в первую очередь, как средство обучения и как первый изучаемый язык программирования. То же самое твердят и современные преподы по информатике, не правда ли? =). Хотя, надо сказать, теперь на эту роль больше отводится Паскаль, но об этом позже. В конце 1975 г. был создан первый интерпретатор Бейсика, который состоял примерно из 20 страниц восьмеричного кода. Его создали два программиста-любителя Дик Уиппл и Джон Арнольд. В этом же году в историю мирового компьютерного программирования вмешивается всеми уважаемый и ругаемый Билл Гейтс =), написавший вместе со своим другом Полом Алленом одну из версий Бейсика. В дальнейшем этот язык программирования зарекомендовал себя как действительно легкий для восприятия новичками и обладающий многими возможностями и для профессионалов. Было создано несколько мощных реализаций BASIC, поддерживающих самые современные концепции программирования, отличающиеся наглядностью и простотой, например, всем известный Microsoft Visual Basic. Pascal и Delphi Паскаль, как всем известно, назван в честь французского математика, разработан он был профессором информатики Николасом Виртом в 1970 году. Очень быстро он получил
очень широкое применение и распространение, потеснив позиции даже Бейсика, его преимущества стали очевидны: относительная легкость изучения, простой и наглядный код программ и очень широкая сфера применения. Почему я назвал его ключевым? Да потому, что на его основе в дальнейшем были образованы не только такие языки, как Ада и Модула2, но и многие другие языки программирования содержат во многом аналогичные Паскалю структуры. Спустя какое-то время Паскаль стараниями Андерса Хейлсберга превратился в мощную профессиональную систему программирования - Turbo Pascal. Эта система стала настолько популярной во всем мире, что выходят несколько ее версий вплоть до 4.0. Именно с этой версии он становится не только быстрым компилятором, но и обладает возможностью разбиения программы на части, компилируемые по отдельности. К осени 1988 года появилась версия 5.0 с еще более развитым программным окружением. Вот тут-то возможности Паскаля показали себя в полной мере: Borland Pascal и используемый в Delphi язык Object Pascal основываются на Turbo Pascal и развивают его идеи. Дельфи по сути является компилятором

языка Pascal. Первая версия Delphi стала первым инструментом для разработки Windows-приложений, объединившая в себе надежный компилятор, визуальную среду для программиста и очень мощные возможности языка по работе с базами данных. Спустя год после выхода Delphi 1 был разработан Dephi 2 для 32-битной системы. Как и у всех удачных языков, у Дельфи появлялись все новые и новые версии, одна из последних на сегодняшний день - 6.0. Таким образом, Delphi является очередным шагом в эволюции компиляторов Паскаля. C и C++ В 1972 году в фирме Bell Labs стараниями Д. Ритчи был разработан язык программирования - Си. Первоначально он создавался как язык для разработки Unix-систем. Си часто называют "переносимым ассемблером", так как он позволял и позволяет работать с данными практически так же эффективно, как и при работе с ассемблером. Именно этим и объясняется его популярность и поныне. В 1977 г. началась работа по созданию машиннонезависимой версии транслятора с языка Си с целью облегчения его переноса в новые окружения. Результатом этого явилось появление совместимых по входному языку трансляторов с Си для 15 различных типов компьютеров. Си обладает двойственностью: с одной стороны, это сравнительно низкоуровневый язык, позволяющий учитывать каждую деталь алгоритма для достижения максимальной производительности. С другой - он и высокоуровневый язык, который учитывает в полной мере особенности архитектуры компьютера, что, в конечном счете, повышает эффективность программирования. Первые версии языка Си ++ были разработаны сотрудником все той же компании Bell Labs Бьярном Страуструпом. Произошло это в 1986 году. Он добавил в язык С объектно-ориентированные черты и исправил некоторые ошибки и неудачные решения предыдущей версии языка. Первыми его пользователями стали сами сотрудники Bell Labs. В 1993-м впервые был реализован коммерческий транслятор, и сам язык был назван "C++". Первым транслятором языка был препроцессор cfront, транслирующий программу на Си++ в эквивалентную программу на Си. Именно с этого времени появляются многочисленные книги по Си ++, и он получает очень широкое распространение. Могу сказать, что C++ продолжает развиваться и совершенствоваться даже в наше время. Сейчас он считается основой для разработки современных больших и сложных проектов. Конечно, как и у всего существующего в природе, у него имеются и слабые стороны, вытекающие из требований эффективности. Java В 1995 году в корпорации Sun Microsystems Кеном Арнольдом и Джеймсом Гослингом был создан такой язык, как Java (не путать с JavaScript!). Согласно преданиям =), технология Java пришла из проекта Oak, основной целью которого была разработка объектно-ориентированных средств описания. Но этот проект в скором времени постигла крупная неудача, и все силы было решено направить целиком на продукты, ориентированные на применение в Инете. Поэтому с апреля 1995 года по сети стала распространяется HotJava - интерфейс просмотра Web-страниц для платформ Sun. В настоящее время HotJava
реализована не только для SunOS и Solaris, но и для многих других Unix-осей, также для Windows NT. Java взяла у Си его синтаксис и в то же время была избавлена от кое-каких неудобных черт последнего. Основное отличие Джавы от Си в том, что для Java происходит компиляция в код некоей абстрактной машины, для которой затем пишется эмулятор (Java Virtual Machine). Этот эмулятор - для вполне реальных систем. Главными отличиями Javaпрограмм, которые называются Java-applications, является использование библиотеки Javaклассов, которые обеспечивают разработку безопасных, распределенных систем. Ada и Ada 95 Язык Ada необычен тем, что он был разработан при поддержке... Минобороны США. Но обо всем по порядку. Итак, в 1978 году был объявлен конкурс на язык программирования, который мог бы использовать неоднородные структуры, умел разделять память и мог бы реализовывать неявные функции преобразования типов. Естественно, победитель нашелся, и случилось это через два года, а именно в 1980 году, когда появились спецификации языка Ада. Язык этот, кроме всего прочего, необычен тем, что разработан он был международным составом программистов компании Honeywell. Но, конечно, и среди них должен быть кто-то

первый и выделяющийся. Ею оказалась Августа Ада Лайвейс, которая была первой программисткой (и кто после этого скажет, что женщине место только на кухне? =), в честь нее и был назван язык-Ада. Наконец, окончательная спецификация языка Ада появилась в феврале 1983 года. В дальнейшем она послужила базисом не только для стандарта Минобороны США, но и стандарта Американского национального института стандартов (ANSI). Структура Ады очень похожа на Паскаль, поэтому этот язык программирования и в данное время считается весьма неплохим средством для разработки больших программных комплексов. Завершающим шагом была выработка все тем же министерством нового стандарта языка: ISO/les 8652:1985 (E), который описывает версию языка Ada95. Примечательно, что именно эта версия стала первой в мире объектно-ориентированной системой программирования, которая имела международный стандарт. Но все-таки оба эти языка (Ada и Ada95) не получили широкого распространения вне военных и проектов крупного масштаба. На этом наш разговор об истории и развитии важнейших языков программирования считаю оконченным. Буду считать свою задачу выполненной, если после прочтения этой статьи ты почерпнул что-то новое для себя и понял, каких усилий стоило некоторым выдающимся людям разработать тот язык программирования, который каждый выбирает для себя.