Информатика и программирование

	1 семестр	2 семестр	3 семестр
Лекции (часов)	16	16	16
Практические занятия (часов)	32	32	32 (48)
Контрольное мероприятие	зачет	зачет	дифференцированный зачет

Парадигма программирования

Парадигма программирования — это система идей и понятий, определяющих стиль написания компьютерных программ. Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Существующие парадигмы программирования:

- **императивное программирование** – описывает процесс вычисления в виде инструкций, изменяющих состояние программы;
- **декларативное программирование** – программа генерируется по ее описанию (HTML-страница);
- **структурное программирование** – представление программы в виде иерархической структуры блоков;
- **функциональное программирование** – заключается в выполнении ряда функций;
- **объектно-ориентированное программирование** – основными концепциями являются понятия объектов и классов.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) позволяет разложить проблему на составные части, каждая из которых становится самостоятельным **объектом**. Каждый из объектов содержит свой собственный код и данные, которые относятся к этому объекту.

Любая программа, написанная на языке ООП, отражает в своих данных состояние физических предметов либо абстрактных понятий – **объектов программирования**, для работы, с которыми она предназначена.

Все данные об объекте программирования и его связях с другими объектами можно объединить в одну структурированную переменную. Описание множества однотипных объектов называется **классом**.

С объектом связывается набор действий, иначе называемых **методами**. С точки зрения языка программирования набор действий или методов – это функции, получающие в качестве обязательного параметра указатель на объект и выполняющие определенные действия с данными объекта программирования. Технология ООП запрещает работать с объектом иначе, чем через методы, таким образом, внутренняя структура объекта скрыта от внешнего пользователя.

Объектно-ориентированные языки программирования

Неполный список объектно-ориентированных языков программирования:

- C#
- C++
- F#
- Java
- Delphi
- Eiffel
- Simula
- D
- Io
- Objective-C
- JavaScript
- JScript .NET
- Ruby
- Smalltalk
- Ada
- Xbase++
- X++
- Vala
- PHP
- Ceylon
- Object Pascal
- VB.NET
- Visual DataFlex
- Perl
- PowerBuilder
- Python
- Scala
- ActionScript (3.0)

История C++

Язык C++ возник в начале 1980-х годов, когда сотрудник фирмы Bell Labs **Бьёрн Страуструп** придумал ряд усовершенствований к языку C под собственные нужды. Когда в конце 1970-х годов Страуструп начал работать в Bell Labs над задачами теории очередей (в приложении к моделированию телефонных вызовов), он обнаружил, что попытки применения существующих в то время языков моделирования оказываются неэффективными, а применение высокоэффективных машинных языков слишком сложно из-за их ограниченной выразительности. Так, язык **Симула** имеет такие возможности, которые были бы очень полезны для разработки объемного программного обеспечения, но работает слишком медленно, а язык BCPL достаточно быстр, но слишком близок к языкам низкого уровня и не подходит для разработки объемного программного обеспечения.

Страуструп дополнил язык C возможностями работы с классами и объектами. В результате практические задачи моделирования оказались доступными для решения как с точки зрения времени разработки (благодаря использованию Симула-подобных классов), так и с точки зрения времени вычислений (благодаря быстродействию C).

Цели создания языка C++

При создании C++ Бьёрн Страуструп ставил цели:

- Получить универсальный язык со статическими типами данных, эффективностью и переносимостью языка C.
- Непосредственно и всесторонне поддерживать множество стилей программирования, в том числе процедурное программирование, абстракцию данных, объектно-ориентированное программирование и обобщённое программирование.
- Дать программисту свободу выбора, даже если это даст ему возможность выбирать неправильно.
- Максимально сохранить совместимость с C: любая конструкция, допустимая в обоих языках, должна в каждом из них обозначать одно и то же и приводить к одному и тому же поведению программы.
- Избегать особенностей, которые зависят от платформы или не являются универсальными.
- «Не платить за то, что не используется» — неиспользуемые языковые средства не должны приводить к снижению производительности программ.
- Не требовать сложной среды программирования.

Понятия объекта и класса

Объект – это структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии.

Класс – это описание множества объектов программирования (объектов) и выполняемых над ними действий.

Класс можно сравнить с чертежом, согласно которому создаются объекты. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области решаемой задачи.

```
struct myclass {          /*класс - описание множества объектов*/
    int   data1;
    ...
};
void  method1(struct myclass *this,...)   /*метод*/
    { ... this->data1 ... }
void  method2(struct myclass *this,...)   /*метод*/
    { ... this->data1 ... }
...
struct myclass obj1, obj2;            /*объекты*/
... method1(&obj1,...);               /*для работы с объектами*/
... method2(&obj2,...);          /*применяются методы*/
```

Понятия объекта и класса

Объект – это структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии.

Класс – это описание множества объектов программирования (объектов) и выполняемых над ними действий.

Класс можно сравнить с чертежом, согласно которому создаются объекты. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области решаемой задачи.

Создание объекта на базе класса называется **инстанцированием** (Instance – сущность).

Основные понятия ООП

Любая функция в программе представляет собой метод для объекта некоторого класса.

Класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых объектов программирования.

Каждый новый шаг в разработке алгоритма должен представлять собой разработку нового класса на основе уже существующих.

Вся программа в таком виде представляет собой объект некоторого класса с единственным методом run (выполнить).

Программирование «от класса к классу» включает в себя ряд новых понятий. Основными понятиями ООП являются

- **инкапсуляция**,
- **наследование,**
- **полиморфизм**.

Основные понятия ООП

Любая функция в программе представляет собой метод для объекта некоторого класса.

Класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых объектов программирования.

Каждый новый шаг в разработке алгоритма должен представлять собой разработку нового класса на основе уже существующих.

Вся программа в таком виде представляет собой объект некоторого класса с единственным методом run (выполнить).

Программирование «от класса к классу» включает в себя ряд новых понятий. Основными понятиями ООП являются

- **инкапсуляция**,
- **наследование,**
- **полиморфизм**.

Основные понятия ООП 18+

Инкапсуляция данных

Инкапсуляция данных – это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В ООП код и данные могут быть объединены вместе (в так называемый «черный ящик») при создании объекта.

Внутри объекта коды и данные могут быть закрытыми или открытыми.

Закрытые коды или данные доступны только для других частей того же самого объекта и, соответственно, недоступны для тех частей программы, которые существуют вне объекта.

Открытые коды и данные, напротив, доступны для всех частей программы, в том числе и для других частей того же самого объекта.

Капсула: основное содержимое скрыто от пользователя, а доступна только внешняя оболочка.

Наследование

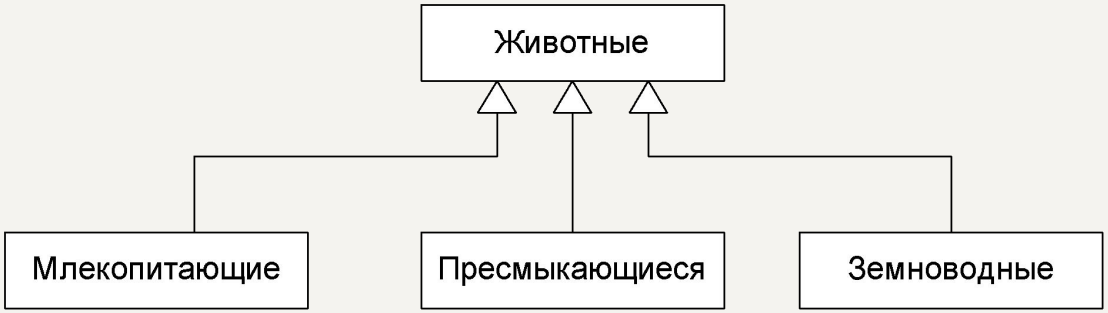
Наследование. Новый, или **производный** класс может быть определен на основе уже имеющегося, или **базового** класса.

При этом новый класс сохраняет все свойства старого: данные объекта (**поля**) базового класса включаются в данные объекта (поля) производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса.

Иначе говоря, новый класс **наследует** как поля старого класса, так и методы их обработки.

Кроме того, наследуемый класс может содержать собственные поля и методы, которые доступны в нем наряду с полями и методами базового класса.

Наследование



Если объект наследует свои свойства от одного родителя, то говорят об **одиночном наследовании**.

Если объект наследует данные и методы от нескольких базовых классов, то говорят о **множественном наследовании**.

Пример наследования – определение структуры, отдельный член которой является ранее определенной структурой.

Полиморфизм

Полиморфизм – это свойство, которое позволяет один и тот же идентификатор (одно и то же имя) использовать для решения двух и более схожих, но технически разных задач.

Целью полиморфизма, применительно к ООП, является использование одного имени для задания действий, общих для ряда классов объектов. Такой полиморфизм основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции).

Будучи доступным в некоторой точке программы, объект , даже при отсутствии полной информации о его типе, всегда может корректно вызвать свойственные ему методы.

Полиморфная функция – это семейство функций с одним и тем же именем, но выполняющие различные действия в зависимости от условий вызова.

Например, нахождение абсолютной величины в языке С требует трех разных функций для разных типов данные и выполняет одни и те же действия:

```
int abs(int);
long labs(long);
double fabs(double);
```

Расширение языка Си

Все основные операции, операторы, типы данных языка С присутствуют в С++. Некоторые из них усовершенствованы и добавлены принципиально новые конструкции, которые и позволяют говорить о С++ как о новом языке, а не просто о новой версии языка.

В языке Си для комментариев используются символы

/* - начало комментария;

*/ - конец коментария.

Вся последовательность, заключенная между этими символами, является комментарием.

Это наиболее удобно для написания многострочных комментариев, и является единственным видом комментариев в языке С.

```
int a;   /* целая переменная */
```

В дополнение к этому, для написания коротких комментариев, в языке С++ используются символы //

При этом комментарием является все, что расположено после символов // и до конца строки:

```
float b; // вещественная переменная
```

Переменные C++

Язык C++ является блочно сконструированным.

Блок – составной оператор, заключенный внутрь символов `{...}`.

Для переменных в C++ справедливо следующее.

- Переменные, описанные внутри блока, вне блока недоступны.
- Переменные внешнего блока доступны во внутреннем, если он их не переопределяет.
- Распределение памяти для переменной происходит при входе в блок, и переменная перестает существовать при выходе из блока.
- Описание переменных может находиться в любом месте блока, в отличии от C, где все объявления находились в начале блока.
- Время жизни переменной – от ее описания до конца блока.

C

```
int i;
...
for(i=0;i<5;i++)
{  тело цикла; }
```

C++

```
for(int i=0;i<5;i++)
{  тело цикла; }
```

Константы C++

Константы в C++ аналогичны константам в C. Но для представления константы в C использовалась только директива препроцессора `#define`.

```
#define MAX 100
```

Объявление константы в C++ аналогично объявлению переменной, но обязательно с начальным значением и ключевым словом `const`:

```
const тип имя = НачальноеЗначение;
const int n=10;
```

Область видимости константы такая же, как у обычной переменной.

С помощью ключевого слова `const` можно объявить указатель на константу

```
const тип *имя;
const int *m; // m – указатель на константу типа int
const int n=3;
m = &n;
```

Константы C++

Еще одна возможность const состоит в возможности создавать постоянный указатель на величину указанного типа

тип *const **ИмяПеременной = Значение;**

```
int i;
int *const ptri=&i;
```

Использование const имеет несколько преимуществ по сравнению с #define.

- При объявлении константы с использованием const явно указывается тип величины.
- Константа, объявленная с использованием const, просто согласуется с производными типами, например, объявление массива:

```
const int base_vals[5] = { 1000, 2000, 3500, 6000, 10000};
```

- Идентификаторы const подчиняются тем же правилам, что и переменные. Можно создавать константы с различной областью видимости.

Константы C++

Недостатки использования `const` по сравнению с `#define`.

C

```
#define SIZE 5
...
int mas[SIZE];
```

C++

```
const int SIZE=5;
...
int mas[SIZE]; // ошибка
```

Перечислимый тип C++

С помощью ключевого слова enum можно объявить особый целочисленный тип с набором именованных целых констант, называемых перечислимыми константами:

```
enum тег {СписокИменованныхКонстант};
enum day {sun,mon,tue,wen,thu,fri,sat};
enum flag {false,true};
```

С помощью такого определения создается целочисленный тип day с названиями 7 дней недели, каждое из которых является целочисленной константой.

Перечислимые константы — идентификаторы, которые по умолчанию имеют следующие значения: 0, 1, 2, 3, 4, 5, 6. Первому присваивается значение 0, и каждому последующему – на 1 больше предыдущего.

Перечислимый тип C++

Если не устраивают значения по умолчанию, то перечислимые константы могут быть инициализированы произвольными целыми константами или константными выражениями

```
enum number {a=54,b,c=60,d=c+5}; // b=55, d=65
```

Перечислимая константа может быть объявлена анонимно (без тега):

```
enum {off,on} signal;
signal=on;
```

Поточный ввод-вывод в C++

В C++, как и в C, нет встроенных в язык средств ввода-вывода.
В C для этих целей используется библиотека `stdio.h`.

В C++ разработана новая библиотека ввода-вывода (iostream), использующая концепцию объектно-ориентированного программирования:

```
#include <iostream>
```

Библиотека `iostream` определяет три стандартных потока:

- `cin` стандартный входной поток (`stdin` в C)
- `cout` стандартный выходной поток (`stdout` в C)
- `cerr` стандартный поток вывода сообщений об ошибках (`stderr` в C)

Для их использования в Microsoft Visual Studio необходимо прописать строку:

```
using namespace std;
```

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

- `>>` получить из входного потока
- `<<` поместить в выходной поток

Поточный ввод-вывод в C++

Ввод значения переменной:

```
cin >> идентификатор;
```

При этом из входного потока читается последовательность символов до пробела, затем эта последовательность преобразуется к типу «идентификатора», и получаемое значение помещается в «идентификатор»:

```
int n;
cin >> n;
```

Возможно многократное назначение потоков:

```
cin >> переменная1 >> переменная2 >>...>> переменнаяn;
```

При наборе данных на клавиатуре значения для такого оператора должны быть разделены символами (пробел, \n, \t).

```
int n;
char j;
cin >> n >> j;
```

Поточный ввод-вывод в C++

Вывод информации:

```
cout << значение;
```

Здесь «значение» преобразуется в последовательность символов и выводится в выходной поток:

```
cout << n;
```

Возможно многократное назначение потоков:

```
cout << 'значение1' << 'значение2' << ... << 'значение n';
int n;
char j;
cin >> n >> j;
cout << "Значение n равно" << n << "j=" << j;
```

Манипуляторы потока в C++

Функцию - манипулятор потока можно включать в операции помещения в поток и извлечения из потока (<<, >>).

Рассмотрим основные манипуляторы C++:

Манипулятор	Описание
`endl`	Помещение в выходной поток символа конца строки '\n'
`dec`	Установка основания 10-ой системы счисления
`oct`	Установка основания 8-ой системы счисления
`hex`	Установка основания 16-ой системы счисления
`setbase`	Вывод базовой системы счисления
`width(ширина)`	Устанавливает ширину поля вывода

Манипуляторы потока в C++

Манипулятор	Описание
`fill('символ')`	Заполняет пустые знакоместа значением символа
`precision (точность)`	Устанавливает количество значащих цифр в числе (или после запятой) в зависимости от использования fixed
`fixed`	Показывает, что установленная точность относится к количеству знаков после запятой
`showpos`	Показывает знак + для положительных чисел
`scientific`	Выводит число в экспоненциальной форме
`get()`	Ожидает ввода символа

Сравнение ввода-вывода в C и C++

C++

```
#include <iostream>
using namespace std;
void main()
{    int n;
     cout << "Введите n:";
     cin >> n;
     cout << "Значение n равно: " << n << endl;
     cin.get(); cin.get();
}
```

C

```
#include <stdio.h>
void main()
{    int n;
     printf("Введите n:");
     scanf("%d",&n);
     printf("Значение n равно: %d\n",n);
     getchar(); getchar();
}
```

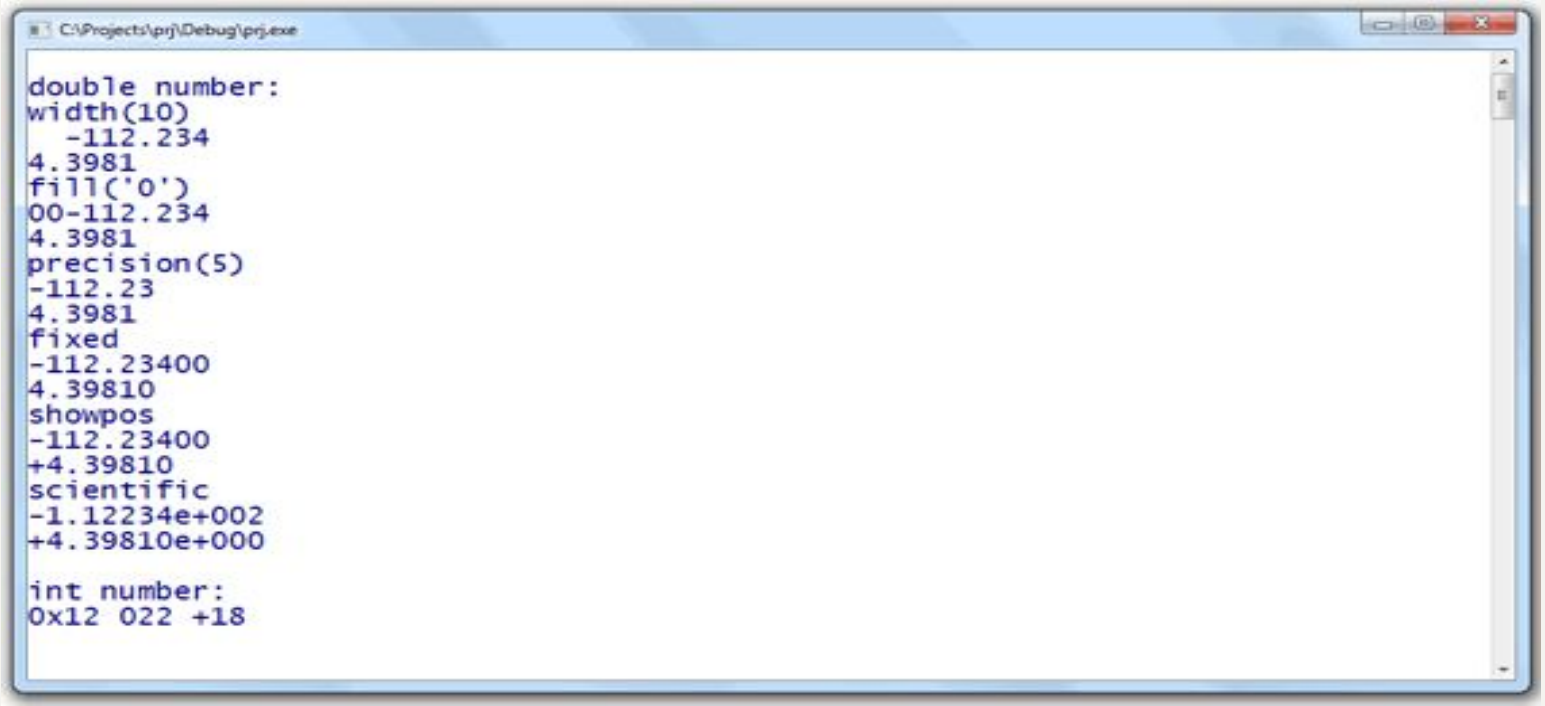
Манипуляторы потока в C++

```
#include <iostream>
using namespace std;
int main() {
  double a = -112.234, b = 4.3981;
  int c = 18;
  cout << endl << "double number:" << endl;
  cout <<"width(10)" << endl;
  cout.width(10);
  cout << a << endl << b << endl;
  cout <<"fill('0')" << endl;
  cout.fill('0');
  cout.width(10);
  cout << a << endl << b << endl;
```

Манипуляторы потока в C++

```
  cout <<"precision(5)" << endl << a << endl << b << endl;
  cout << "fixed" << endl << fixed << a << endl << b << endl;
  cout << "showpos" << endl << showpos << a << endl << b << endl;
  cout << "scientific" <<endl << scientific << a << endl << b << endl;
  cout << endl << "int number:" << endl;
  cout << showbase << hex << c <<" " << showbase << oct << c << " ";
  cout << showbase << dec << c << endl;
  cin.get();
  return 0;
}
```

Манипуляторы потока в C++



Использование void

Ключевое слово void в стандарте языка C используется для указания того, что функция не возвращает значения и не принимает параметров:

```
void main(void)  {...}
```

В C++ введены еще 2 способа использования void:

- в операциях приведения типа для указания компилятору, что значения вычисленного выражения игнорируются:

```
a = (void) func(n);
```

- Объявление указателя на неопределенный тип:

```
void *ptr;
```

Использование void

```
void *ptr;
```

Такому указателю может быть присвоен указатель на любой тип, но не наоборот

```
void *ptr;            // Указатель на void
int i, *ptri;         // Целая переменная, указатель на int
ptr= &i;          // Допустимо
ptr= ptri;            // Допустимо
ptri= (int)ptr;       // Допустимо
// ptri=ptr;          // Недопустимо
```

Для последней операции необходимо явное приведение типа.

Над указателем неопределенного типа нельзя выполнять операцию разыменования без явного приведения типа.

Ссылки

В C++ введен новый тип данных – **ссылка**. Ссылка позволяет определять альтернативное имя переменной.

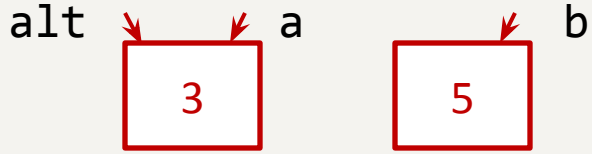
Формат объявления ссылки:

тип &идентификатор_1 = идентификатор_2;

Такое объявление назначает переменной с именем **идентификатор_2** второе имя **идентификатор_1**.

Ссылка при объявлении всегда должна быть проинициализирована!

```
int a = 3,b = 5;
int &alt=a; // alt – другое имя переменной a (ссылка на a)
alt = b;// a=b;
alt++;  // a++;
```



Ссылки

Если объявлен указатель

```
int *ptr = &a;
```

то истины следующие выражения:

```
*ptr == alt;     // истина
ptr == &alt;     // истина
```

Ссылку можно рассматривать как постоянный указатель, который всегда разыменован, для него не надо выполнять операцию косвенной адресации *.

Ссылка не создает копии объекта, а является лишь другим именем объекта.

Возможно инициализировать ссылка на константу:

```
const char &new_line='\n';
```

В этом случае компилятор создает некоторую временную переменную temp и ссылку на нее:

```
char temp = '\n';
const char &new_line = temp;
```

Ссылки

Основной причиной введения в C++ нового типа данных – ссылки явилась необходимость передачи параметров в функцию через ссылку и получение возвращаемого значения в виде ссылки. Это используется в двух случаях:

- для передачи в функцию больших структур, чтобы избежать копирования аргументов в стек;
- для передачи функции аргументов, которые должны быть изменены самой функцией.

В обоих случаях можно использовать указатели, но это влечет за собой дополнительные расходы:

- во-первых, в функции для данного параметра надо выполнять операцию разыменования,
- во-вторых, при вызове функции надо передавать не саму переменную, а ее адрес.

Ссылки

Пример Функция, меняющая местами два целых числа:

Без использования ссылок

```
void swap (int *a, int *b)
{
 int temp = *a;
 *a = *b;
 *b = temp;
}
```

Вызов функции

```
int x = 10;
int y = 5;
swap(&x, &y);
```

С использованием ссылок:

```
void swap (int &a, int &b)
{
 int temp = a;
 a = b;
 b = temp;
}
```

Вызов функции

```
int x = 10;
int y = 5;
swap(x, y);
```

Ссылки

При использовании ссылок в качестве параметров, наряду с указанными преимуществами есть два существенных недостатка:

- Фактический аргумент, переданный в функцию по ссылке, может быть изменен функцией без ведома вызывающей программы. Чтобы этого избежать, параметры, которые не должны изменяться, должны определяться с ключевым словом const. При попытке изменить параметр, объявленный как const будет сообщение об ошибке.
- Если при вызове функции происходит несоответствие типов фактических и формальных параметров, C++ выполняет преобразование типа, но для ссылок преобразование типа выполняется через создание промежуточной переменной.

Ссылки

```
void swap (int &, int &);
main()
{
 int x=10;
 unsigned int y;
 y=5;
swap(x, y);
}
```

Компилятор будет выполнять следующие преобразования:

```
int temp = (int)y;
int &t = temp;
swap(x, t);
```

В результате `swap()` поменяет местами значения `x` и `temp`. После выхода из функции переменная temp удалится, а y останется неизменной.

Ссылки

В C++ функции могут не только принимать ссылку в качестве аргумента, но и возвращать ссылку на переменную. Выражение вызова такой функции может появиться в любой части операции присваивания.

При этом необходимо учитывать, что

- если возвращаемое значение — указатель, то нельзя оператором return возвращать адрес локальной переменной.
- если возвращаемое значение — ссылка, то нельзя оператором return возвращать локальную переменную. (Так как после выхода из функции переменная не существует, и мы получим повисшую ссылку).

Чаще всего потребность в ссылках возникает при перегрузке операций.

Функции

Сигнатура функции определяет правила использования функции. Обычно представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

```
#include <iostream>
using namespace std;
int sum(int, int);   // сигнатура
void main()
{
     int a, b;
     cout << "Введите 2 числа:";
     cin >> a >> b;
     int r = sum(a,b);
     cout << "Сумма равна " << r;
     cin.get(); cin.get();
}
int sum(int x, int y)
{ return(x+y);}          // семантика
```

Функции. Прототипы функций

Определение функции в программе выглядит следующим образом:

```
ЗаголовокФункции
{
        ТелоФункции
}
```

Заголовок функции имеет следующий вид:

```
тип ИмяФункции (СпецификацияФормальныхПараметров)
```

Если функция не возвращает значения, то ее тип `void`. Такая функция называется **процедурой**.

При обращении к функции формальные параметры заменяются фактическими, причем соблюдается строгое соответствие параметров по типам.

Функции. Прототипы функций

В отличие от языка С, С++ не предусматривает автоматического преобразования в тех случаях, когда фактические параметры не совпадают по типам с соответствующими им формальными параметрами.

Говорят, что язык С++ обеспечивает «строгий контроль типов».

В связи с этой особенностью языка С++ проверка соответствия типов формальных и фактических параметров выполняется на этапе компиляции.

Строгое согласование по типам между формальными и фактическими параметрами требует, чтобы в модуле до первого обращения к функции было помещено либо ее определение, либо ее описание (прототип), содержащее сведения о ее типе, о типе результата (то есть возвращаемого значения) и о типах всех параметров. Наличие такого прототипа либо полного определения позволяет компилятору выполнять контроль соответствия типов параметров.

Функции. Прототипы функций

Прототип (описание) функции может внешне почти полностью совпадать с заголовком ее определения:

тип ИмяФункции (СпецификацияФормальныхПараметров);

Основное различие – точка с запятой в конце описания (прототипа). Второе отличие – необязательность имен формальных параметров в прототипе даже тогда, когда они есть в заголовке определения функции.

Спецификация формальных параметров – это либо пусто (void), либо список спецификаций отдельных параметров. Спецификация каждого параметра в определении функции имеет вид:

тип ИмяПараметра

тип ИмяПараметра = ЗначениеПоУмолчанию

Для параметра может быть задано (а может отсутствовать) значение по умолчанию. Это значение используется в том случае, если при обращении к функции соответствующий параметр не передан.

Если параметр имеет значение по умолчанию, то все параметры, специфицированные справа от него, также должны иметь значения по умолчанию.

Функции. Значения параметров по умолчанию

Пусть нужно вычислить n в степени k, где k чаще всего равно 2.

```
int pow(int n, int k=2)  // по умолчанию k=2
{
    if( k== 2) return( n*n );
    else return( pow( n, k-1 )*n );
}
```

Вызов функции из тела самой функции называется **рекурсивным вызовом**.

Вызывать эту функции можно двумя способами:

```
t = pow(i);
q = pow(i, 5);
```

Значение по умолчанию может быть задано либо при объявлении функции, либо при определении функции, но только один раз.

Совет: задавать значение по умолчанию в определении функции, поскольку объявлений функции может быть несколько если функция используется в разных файлах.

Функции. Перегрузка функций

При перегрузке функция с одним именем по-разному выполняется и возвращает разные значения при обращении к ней с разными по типам и количеству фактическими параметрами.

Например, функция, возвращающая максимальное значение элементов одномерного массива, передаваемого ей в качестве параметра.

Массивы, используемые как фактические параметры, могут содержать элементы разных типов, но пользователь функции не должен беспокоиться о типе результата. Функция всегда должна возвращать значение того же типа, что и тип массива – фактического параметра.

В этом случае придется написать три разных варианта функции с одним и тем же именем.

```
int max(int *a, int n);
float max(float *a, int n);
double max(double *a, int n);
```

Функции. Перегрузка функций

Для обеспечения перегрузки функций необходимо для каждого имени определить, сколько разных функций связано с ним, т.е. сколько вариантов сигнатур допустимы при обращении к ним.

Распознавание перегруженных функций при вызове выполняется по их сигнатурам. Перегруженные функции, поэтому должны иметь одинаковые имена, но спецификации их параметров должны различаться по количеству и (или) по типам, и (или) по расположению.

При использовании перегруженных функций нужно с осторожностью задавать начальные значения их параметров.

Например, если в одной программе перегружены функции

```
int sum(int a, int b=1)  { return(a+b); }
int sum(int a)           { return(a+a); }
```

то вызов

```
int r = sum(2);  // ошибка
```

выдаст ошибку из-за неоднозначности толкования sum().

Функции. Встраиваемые функции

В базовом языке С директива препроцессора `#define` позволяла использовать макроопределения для записи вызова небольших часто используемых конструкций. Некорректная запись макроопределения может приводить к ошибкам, которые очень трудно найти. Макроопределения не позволяют определять локальные переменные и не выполняют проверки и преобразования аргументов.

Если вместо макроопределения использовать функцию, то это удлиняет объектный код и увеличивает время выполнения программы.

Кроме того, при работе с макроопределениями необходимо тщательно проверять раскрытия макросов:

```
#define SUMMA(a, b) a + b
rez = SUMMA(x, y)*10;
```

После работы препроцессора получим:

```
rez = x + y*10;
```

Функции. Встраиваемые функции

В C++ для определения функции, которая должна встраиваться как макроопределение используется ключевое слово `inline`. Вызов такой функции приводит к встраиванию кода inline-функции в вызывающую программу. Определение такой функции может выглядеть следующим образом:

```
inline double SUMMA(double a, double b)
      {
       return(a + b);
      }
```

При вызове этой функции

```
rez = SUMMA(x,y)*10;
```

будет получен следующий результат:

```
rez=(x+y)*10
```

Функции. Встраиваемые функции

При определении и использовании встраиваемых функций необходимо придерживаться следующих правил:

- Определение и объявление функций должны быть совмещены и располагаться перед первым вызовом встраиваемой функции.
- Имеет смысл определять `inline` только очень небольшие функции, поскольку любая inline-функция увеличивает программный код.
- Различные компиляторы накладывают ограничения на сложность встраиваемых функций. Компилятор сам решает, может ли функция быть встраиваемой. Если функция не может быть встраиваемой, компилятор рассматривает ее как обычную функцию.

Динамическое выделение памяти

В C работать с динамической памятью можно при помощи соответствующих функций распределения памяти (`calloc`, `malloc`, `free`), для чего необходимо подключить библиотеку

```
#include <malloc.h>
```

C++ использует новые методы работы с динамической памятью при помощи операторов `new` и `delete`:

- `new` — для выделения памяти;
- `delete` — для освобождения памяти.

Динамическое выделение памяти

Оператор new используется в следующих формах:

```
new тип;      // для переменных
new тип[размер];     // для массивов
```

Память может быть выделена для одного объекта или для массива любого типа, в том числе типа, определенного пользователем. Результатом выполнения операции new будет указатель на отведенную память, или нулевой указатель в случае ошибки.

```
int *ptr_i;
double *ptr_d;
struct person *human;
.................
ptr_i = new int;
ptr_d = new double[10];
human = new struct person;
```

Динамическое выделение памяти

Память, выделенная в результате выполнения new, будет считаться выделенной до тех пор, пока не будет выполнена операция delete.

Освобождение памяти связано с тем, как выделялась память – для одного элемента или для нескольких. В соответствии с этим существует и две формы применения delete

```
delete  указатель;   // для одного элемента
delete[] указатель;  // для массива
```

Например, для приведенного выше случая, освободить память необходимо следующим образом:

```
delete ptr_i;
delete[] ptr_d;
delete human;
```

Освобождаться с помощью delete может только память, выделенная оператором new.

Пример динамического выделения памяти

```
#include <iostream>
using namespace std;
int main()
{
 int size;
 int *dan;
 cout << "Ввести размерность массива: ";
 cin >> size;
 dan = new int[size];
 for (int i=0; i<size; i++)
 {
   cout << "dan[" << i << "]= ";
   cin >> dan[i];
 }
 delete[] dan;
 cin.get(); cin.get();
 return 0;
}
```

dan – базовый адрес динамически выделяемого массива, число элементов которого равно size. Операцией delete освобождается память, выделенную при помощи new.