Тема 1. Введение в C++ (продолжение)

Потоковый ввод-вывод

Заголовочный файл (объявления) функций стандартной библиотеки ввода/выводв
`#include <cstdio>` — язык C
`#include <iostream>` — язык C++
По сути очень близки, но у каждой есть свои удобства и неудобства.
Концепция:
stdio вызов функций ввода/вывода, параметры задают:
— где выполняется ввод/вывод
— формат (преобразования значений в печатные символы при выводе и символов в значения при вводе)
— значения для вывода и указатели на переменные для ввода
iostream — объект "поток", который отвечает за ввод/вывод
— предоставляет разумную форму ввода/вывода по умолчанию
— можно настраивать форму ввода/вывода
— запись операции как последовательность компонент, передаваемых в поток (извлекаемых из потока).
Консольный ввод/вывод (экран и клавиатура). Ввод/вывод с файлами.
Стандартные каналы (потоки)
C: stdin stdout stderr
C++: std::cin std::cout std::cerr
`using namespace std;` — чтобы не писать `std::cin` и т.п.

```
 Функции stdio              операторы iostream
     scanf                       cin >>
     printf                      cout <<
и другие аналогичные

 Спецификации преобразования stdio   %d %u %x %c %s
                                     %f  %lf  %e  %le %g  %lg
       %N...       ширина                    width()
       %.M         точность                  precision()
       \n                                      endl

управление формой вывода   flags()   setf()   unsetf()
                           width()   precision()
```

```
    double x, y;
    // %15.12f
    cout.width(15);
    cout.precision(12);
    cout << x << " " << y << endl;
    // %15.12e
    cout << scientific << x << " " << scientific << y << endl;
```

Флагов много разных, нужно смотреть описание.
Для работы с файлами используются классы ifstream и ofstream.
Открытие файла — конструктор или open
успешность открытия и также чтения или записи — устанавливается бит состояния потока. Проверки

```
   is_open()
   operator!              // fail()
   operator bool          // !fail()
```

```
   bool good()
.  bool eof()
   bool fail()
   bool bad()
```

Переопределение операций вводв-вывода для своих классов

```
class A;

ostream operator<<(ostream &out, const A &a)
{
    out << ... какие-то значения из a
    return out;
}
// если надо, то объявляется friend в классе
// аналогично для ввода istream
```

Также есть удобный класс stringstream — запись в “строку”.

Концепция RAII

RAII = Resource Acquisition Is Initialization

— Приобретение ресурсов проводится при инициализации.

Если объекту требуются ресурсы (память, файлы и т.п.), то он должен позаботиться об этом в конструкторе и потом освободить эти ресурсы в деструкторе.

Пример — FILEPTR.

```
#include <cstdio>

class FILEPTR {
    FILE *f;
    void operator=(const FILEPTR &) = delete;
    FILEPTR (const FILEPTR &) = delete;
public:
    FILEPTR(const char *fname, const char *mode) { f = fopen(fname, mode); }
    operator FILE*() { return f; }
    operator bool() { return (f != nullptr); }
    ~FILEPTR() { if (f) fclose(f); }
};

void somefun(FILEPTR h)
{
    fprintf(h,"in somefun\n");
}

void otherfun(FILEPTR &h)
{
    fprintf(h,"in otherfun\n");
}

int main()
{
    FILEPTR f("text.txt", "w");    // захватываем ресурс при инициализации
    if (!f) return -1;             // operator bool()
```

```
    fprintf(f, "it works!\n");     // operator FILE*()
    // somefun(f);                 // конструктор копирования запрещен
    otherfun(f);                   // по ссылке передавать в функцию можно
    fprintf(f, "it works again!\n");
    return 0;                      // освобождаем ресурс в деструкторе
}
```

Основная идея — объект захватывает ресурс и управляет им монопольно. Для этого запрещается дублировать возможность доступа и управления этим ресурсом (указатель f) посредством операций присваивания и конструкторов копирования, т.е. эти функции отключаются.

конец лекции 4

Концепция наследования

Наследование — производный класс расширяет возможности базового класса, сохраняя также доступ к старым возможностям базового класса.

Простейший пример наследования.

```
class A {          // базовый класс
   private:
      int xa;
   protected:
      int ya;
   public:
      int za;
      A(int x) : xa(x), ya(x+1), za(x+2) { printf("A: %d\n", x); }
      void Print() { printf("A: %d %d %d\n",xa,ya,za);}
};


class B : public A {      // производный класс
   private:
      int xb;
   protected:
      int yb;
   public:
      int zb;
      B(int x = 0) : A(3), xb(x), yb(x+10), zb(x+20)  { printf("B: %d\n", x); }
      void Print() { printf("B: %d %d   %d %d %d\n",ya,za,xb,yb,zb);}
};

int main()
{
    A a(1);
    a.Print();
    B b(10);
    b.Print();
    return 0;
}
```

Вид наследования:
public — private недоступны, у остальных права как были

protected — private недоступны, у остальных protected
private — private недоступны, у остальных private
Обычно используется public наследование.
Доступ к базовому и порожденным классам от имени объекта
— все функции вызываются от соответствующего класса (т.е. переопределенные, если они есть, или от базового, если не переопределялись и доступны).
Доступ к базовому и порожденным классам по ссылке или указателю. Указатель на базовый класс может показывать на порожденный класс !!!
Невиртуальные функции вызываются в соответствии с объявлениям указателя.
Виртуальные функции вызываются в соответствии с фактическим состоянием указателя (на кого указывает).

```
class A {
   public:
   void f () { printf("A::f()\n"); }
   void g () { printf("A::g()\n"); }
   virtual void h () { printf("A::h()\n"); }
};

class B : public A {
   public:
   void g () { printf("B::g()\n"); }
   void h () { printf("B::h()\n"); }
};

int main()
{
   A a;
   B b;

   printf("от имени класса\n");
   a.f();   // базовый
   a.g();   // базовый
   a.h();   // базовый
   b.f();   // унаследовано от базового
   b.g();   // переопределено в порожденном
   b.h();   // виртуальная и переопределено в порожденном

   printf("через указатели\n");
   A* pa = &a;
   B* pb = &b;
   A* pab = &b;  // разные левая и правая части

   printf("A* p = &A\n");  // все от A
   pa->f();
   pa->g();
   pa->h();
   printf("B* p = &B\n");  // все от B
   pb->f();                // унаследованнные не переопределенные от A
   pb->g();
   pb->h();
   printf("A* p = &B\n");  // унаследованные не виртуальные от A
   pab->f();
   pab->g();
```

```
    pab->h();               // виртуальные от B

    printf("преобразование указателей\n");
    B *qb = dynamic_cast<B*>(pab);
    printf("%p %p\n", (void*)qb, (void*)pab);
    qb->f();
    qb->g();
    qb->h();

    A *qa = dynamic_cast<A*>(pab);
    printf("%p %p\n", (void*)qa, (void*)pab);
    qa->f();
    qa->g();
    qa->h();

    B *qb2 = dynamic_cast<B*>(pa);    // нельзя преобразовать
    printf("%p %p\n", (void*)qb2, (void*)pa);

    return 0;
}
```

Преобразование dynamic_cast позволяет преобразовать указатель на базовый класс в тот указатель, которым на самом деле является указываемй объект. Если же за указателем не стоит такой объект, то преобразование даст 0, что позволяет выяснить что на самом деле стоит за указателем на базовый класс.