Тема: Динамическое связывание

- Наследование и полиморфизм

- Динамическое и статическое связывание
- Наследование – разрешение проблем

Базовый и производный тип

- Если класс ***B*** *публично* произведен из базового класса ***A*** , то по правилам C++ объекта типа ***B*** также может использоваться, как объект типа ***A***. Иными словами, объект производного класса является также объектом базового класса.
- Следовательно
  - объект класса ***B*** может инициализировать объект класса ***A***
  - объект класса ***B*** может быть фактическим аргументом для функции с формальным аргументом класса ***A***
  - то же для возвращаемого значения

Наследование и полиморфизм

Наследование позволяет

- писать полиморфные функции

  (не различать аргументы классов, имеющих общего предка)

- сохранить контроль типов

  (различать аргументы классов, имеющих общего предка)

Родственные классы

```
class Text_Window : public Window {
  public:
     Text_Window(const Point& , const Point& , const String& ) ;
};
class Shell_Window : public Window {
  public:
     Shell_Window(const Point& , const Point& , const String& ) ;
     void change_size(const Point&);
     void execute(const String&);
};
class Input_Window : public Window {
  public:
     Input_Window(const Point& , const Point& , const String& ) ;
     String  read_input();
};
```

Полиморфизм: функция ***blank***

```
#include “window.h”
// функция использует только общие
// для всех видов окон методы
// и является полиморфной
void blank( Window& w ) {
 w.clear();
 w.move_cursor( Point(0,0) );
 w.add( “This window was blank” );
}
```

Использование функции ***blank***

```
#include "window.h"
#include "dos.h" // для sleep
void blank(Window&);
void main( int, char*[ ] ) {
  Text_Window tw(Point(10,2), Point(40,6), "TW");
  Shell_Window sw(Point(10,2), Point(40,6), "SW");
  Input_Window iw(Point(10,2), Point(40,6), "IW");
  sleep(2) ;
  blank(tw) ;
  blank(sw) ;
  blank(iw) ;
  sleep(2) ;
}
```

Использование строгой типизации

```
// функция может работать только с окном вида Input_Window
int yes_or_no(Input_Window& w, const String& question)
{
    String  answer ;
    do {
     w.clear();
     w.move_cursor( Point(0,0) );
     w.add(question);
     answer = w.read_input() ;
    } while ( answer != “yes” && answer != “no” );
    return answer == “yes” ;
}
```

Функция ***collapse***

```
#include “window.h”
// функция очищает и уменьшает окно
// использует только общие
// для всех видов окон методы
// и является полиморфной
void collapse( Window& w ) {
 w.clear();
 w.change_size( Point(5,5) );
}
```

Использование функции ***collapse***

```
#include "window.h"
void collapse(Window&);
void main( int, char*[ ] ) {
 Window w(Point(10,2), Point(40,6), "W");
 Shell_Window sw(Point(10,2), Point(40,6), "SW");
 w.add("This is a text in a Window object") ;
 sw.add("This is a file list in a Shell_Window ") ;
 sw.execute("dir") ;
 sleep(2) ;
 collapse(w) ;
 collapse(sw) ;   // окно недопустимо уменьшится
 sleep(2) ;
}
```

Проблема связывания

- При вызове функции ***collapse*** с фактическим аргументом типа ***Shell_Window*** какой метод будет вызываться?
  - *Window*::*change_size*(**const** *Point*&)
  - *Shell_Window*::*change_size*(**const** *Point*&)
- Требуются средства языка, обеспечивающие второй вызов

Два вида связывания

- ***Статическое*** или ***раннее*** связывание
  - выбор вызываемой функции осуществляется на этапе компиляции
  - определяется по объявленному типу вызывающего объекта
- ***Динамическое*** или ***позднее*** связывание
  - выбор вызываемой функции осуществляется на этапе выполнения
  - определяется по фактическому типу вызывающего объекта

Функция ***collapse***

- При статическом связывании в функции ***collapse*** всегда будет вызываться метод класса ***Window*** согласно объявлению формального аргумента
- При динамическом связывании компилятор откладывает выбор метода до выполнения программы, когда есть точная информация о типе объекта – фактического параметра

Динамическое связывание

- Отвечает принципам ООП
- Многие объектно-ориентированные языки используют только динамическое связывание
- Объект, как экземпляр класса, должен содержать информацию о своем типе
- Требуются накладные расходы по сравнению со статическим связыванием

Выбор вида связывания в C++

- В C++ по умолчанию используется статическое связывание
- Чтобы применить динамическое связывание для конкретного метода, его надо описать ***в базовом классе*** с ключевым словом **virtual**
- Информация о типе объекта будет анализироваться во время выполнения программы и определять вызов метода (т. е. используется *алгоритм вызова*)

Базовый класс ***Window***

```
class Window {
  public:
    Window( const Point& upleft, const Point& size,
            const String& title);
    ~Window( );
    void move(const Point& new_upleft);
    Point upper_left() const;
    Point lower_right() const;
    Point size() const;
    virtual void change_size(const Point& new_size);
    // если новый размер меньше минимального, то окно
    // уменьшается только до минимального размера
```

Как поступать?

- Объявлять методы виртуальными по мере необходимости
  - минус – изменение кода базового класса
  - плюс – нет лишних расходов
- Все методы объявлять виртуальными
  - минус – максимальные накладные расходы
  - плюс – код базового класса не меняется
- Проанализировать возможное развитие класса и объявить виртуальные методы на основе этого анализа
  - **НО** как стать таким умным?

Полиморфные массивы

```
#include “window.h”
#include <dos.h>
void collapse(Window&);
void main( int, char*[ ] ) {
  Window w(Point(10,2),
     Point(40,6), “W”);
  Text_Window tw(Point(20,12),
     Point(20,5), “TW”);
  Shell_Window sw(Point(10,2),
     Point(40,6), “SW”);
  Input_Window iw(Point(10,4),
     Point(10,8), “IW”);
```

```
    Window* all_windows[5];
    all_windows[0] = &w ;
    all_windows[1] = &tw ;
    all_windows[2] = &sw ;
    all_windows[3] = &iw ;
    all_windows[4] = NULL ;
    sleep(2) ;
for( int i=0; all_windows[i]; i++ )
    collapse(*all_windows[i]) ;
    sleep(2) ;
}
```

Копирование и присваивание

```
#include “window.h”
void collapse2( Window w ) {
  w.clear();
  w.change_size( Point(5,5) );
}
void main( int, char*[ ] ) {
  Window w(Point(10,2), Point(40,6), “W”);
  Shell_Window sw(Point(1,2), Point(40,6), “SW”);
  w = sw ;
  collapse2(sw) ;   // окно недопустимо уменьшится
  sleep(2) ;
}
```

Проблема деструктора при наследовании

```
class A {
    public:
      A( );
     ~A( );
    ……..
};
```

```
class B : public A {
    public:
      B( );
     ~B( );
    ……..
};
```

```
void main( int, char*[ ] ) {
    A* po = new B ;     // вызывается конструктор
                        // класса B
    delete po ;     // вызывается деструктор класса A
}
```

Виртуальный деструктор базового класса

```
class A {
    public:
      A( );
     virtual ~A( );
    ……..
};
```

```
class B : public A {
    public:
      B( );
     ~B( );
    ……..
};
```

```
void main( int, char*[ ] ) {
    A* po = new B ;     // вызывается конструктор
                        // класса B
    delete po ;     // вызывается деструктор класса B
}
```

РЕЗЮМЕ

- Наследование и динамическое связывание позволяет
  - добавлять новые производные классы не меняя существующий код
  - писать полиморфные функции (функция ***blank***)
  - писать функции только для определенного производного класса (функция ***yes_or_no***)
- Динамическое связывание требует накладных расходов
- Имеет тонкости применения
  - передача аргумента по ссылке
  - использование указателей