# Лекция 2. Структуры и классы

# Зачем группировать данные?

Какая должна быть сигнатура у функции, которая вычисляет длину отрезка на плоскости?

```
double length(double x1, double y1, double x2, double y2);
```

А сигнатура функции, проверяющей пересечение отрезков?

```
double intersects(double x11, double y11, double x12, double y12,
                  double x21, double y21, double x22, double y22,
                  double *xi, double *yi);
```

Координаты точек являются логически связанными данными, которые всегда передаются вместе.

Аналогично связаны координаты точек отрезка.

# Структуры

*Структура* – способ синтаксически (и физически) сгруппировать логически связанные данные.

```
struct Point {
   double x;
   double y;
}; <- обязательно использование ;

struct Segment {
   Point p1;
   Point p2;
};

double length(Segment segment);

bool intersects(Segment first, Segment second, Point *iPoint);
```

# Объявление структуры

*Структура* – группа связанных переменных (составной тип данных).

*Член структуры* – переменная, которая является частью структуры.

*Имя структуры* – спецификатор пользовательского типа.

```
struct Point2D { <- можно ли назвать 2DPoint?
   double x;
   double y;
} zero; <- объявление переменной


struct Point2 { <- совместима с Point2D?
   double x;
   double y;
};
```

# Инициализация структуры

Перекрытие имен во вложенных областях видимости.

Допустимость типов с одинаковыми именами в одной области.

```
struct Point2D {
    double x;
    double y;
} zero;

int main() {
    struct Point3D {
        double x;
        double y;
        double z;
    } zero;
    cout << sizeof(zero) << endl;
    cout << double.x << endl;
};
```

# Инициализация структуры

```
int main() {
    Point2D first;
    first.x = 0.5;
    first.y = 1.0;

    Point2D second = {2.0, 3.0};

    struct {
        int count;
        char message[13];  <- Использование массива в структуре
    } single = {10, «Hello world!»};

    cout << single.message << endl;
};
```

# Массивы структур

Структура объявляет новый тип данных – можно использовать массивы этого типа.

```
struct Point2D {
    double x;
    double y;
};

struct Segment {
    Point2D points[2];
};


cout << sizeof(Segment) << endl;  <- Каков будет результат?
```

# Указатели на структуры

По аналогии с массивами, можно объявлять указатели на структуры.

```
struct Node {
    struct Holder {   <- Вложенное объявление структуры
        int value;
    };
    Holder holder;
    Node *next;
};

int main() {
    Node second = {20, NULL};
    Node first = {10, &second};

    cout << first.next->holder.value << endl;
};
```

# Передача структур в функцию (по значению)

Структуры передаются в функцию *по значению*.

```
void increment(Point2D point) {
    point.x++;
    point.y++;
}

int main() {
    Point2D point = {0, 0};
    increment(point);
    cout << point.x << " " << point.y << endl;
};
```

# Передача структур в функцию (по ссылке)

Для объектов структур имеется возможность передачи параметра функции *по ссылке*.

```
void shift(Point2D &point, double dx, double dy) {
    point.x += dx;
    point.y += dy;
}

int main() {
    Point2D point = {0, 0};
    shift(point, 1, 2);
    cout << point.x << " " << point.y << endl;
};
```

# Объединение

Состоит из нескольких переменных, которые разделяют одну и ту же область памяти.

Обеспечивает низкоуровневую поддержку принципов полиморфизма.

```
union Integer {
    int value;
    short half[2];
};

int main() {
    Integer integer;
    integer.value = 0xFFFF0000;
    cout << integer.half[0] << " " << integer.half[1] << endl;
};
```

# Класс

Определяет новый тип данных, который задает формат объекта.

Является логической абстракцией.

Включает как данные так и код, предназначенный для выполнения над этими данными.

Связывает данные с кодом – выполняет *инкапсулирование* .

Функции и переменные, входящие в класс называются его *членами:*

- Член данных (поле, атрибут)
- Функция-член (метод)

# Объявление класса

Создается с помощью ключевого слова *class.*

Объявление синтаксически подобно определению структуры.

```
class People {
    int age;     <- По умолчанию private
    string name;
public:    <- Модификатор доступа
    int getAge();
    string getName();
};

int main() {
    People people;
    cout << sizeof(People) << endl;
    cout << sizeof(people) << endl;
};
```

# Определение функций класса

Осуществляется с указанием класса, которому принадлежит функция

```
class People {
    int age;
    string name;
public:      <- Публичный интерфейс класса
    int getAge();
    string getName();
};
             <- Оператор разрешения видимости
int People::getAge() {
    return age;
}

string People::getName() {
    return name;
}
```

# Инвариант класса

*Публичный интерфейс* – набор методов, доступный внешним пользователям класса.

*Инвариант класса* – утверждение, которое (должно быть) истинно применительно к любому объекту данного класса в любой момент времени (за исключением переходных процессов в методах объекта).

Для сохранения инвариантов класса:

- Все поля должны быть закрытыми
- Публичные методы должны сохранять инварианты класса

Закрытие полей позволяет абстрагироваться от способа хранения данных объекта.

# Модификаторы доступа

- Public – доступ открыт всем, кто видит определение данного класса;
- Protected – доступ открыт классам, производным от данного;
- Private – доступ открыт самому классу (т.е. функциям-членам данного класса) и друзьям (friend) данного класса, как функциям, так и классам.

По умолчанию все функции и поля класса объявлены закрытыми.

Поля рекомендуется делать закрытыми (private) и предоставлять доступ к ним через get*Value* и set*Value* методы.

# Встраиваемые функции (inline)

Небольшая по объему функция, код которой подставляется в место её вызова.

```
class People {
    int age;
    string name;
public:
    int getAge() {
        return age;
    }
    string getName();
};

inline string People::getName() {
    return name;
}
```

# Неявный указатель this

В каждой функции класса имеется указатель на объект, через который данная функция вызывается.

```
void People::setAge(int newAge) {
    age = newAge;
}

/* С неявным указателем this */
void People::setAge(/* People *this */, int newAge) {
    age = newAge;
}

/* С использованием указателя this */
void People::setAge(int age) {
    this->age = age;
}
```

# Перегрузка функций

Определение нескольких функций с одинаковым именем, но различными параметрами.

```
class Point2D {
   int x;
   int y;
public:
   void move(int dx, int dy);
   void move(Point2D vector);
} zero;

void move(Point2D vector);

int main() {
   move(zero);
};
```

# Абстракция и инкапсуляция

```
class IntArray2D {
   int width;
   int height;
   int *data;
public:
   int &get(int row, int column) { return data[row * width + column]; }
   int getWidth() { return width; }
   int getHeight() { return height; }
};

IntArray2D createArray();

int main() {
   IntArray2D array = createArray();
   for (int row = 0; row < array.getHeight(); ++row) {
      for (int column = 0; column < array.getWidth(); ++column) {
         cout << array.get(row, column) << " ";
      }
      cout << endl;
   }
};
```

# Определение констант

Ключевое слово `const` позволяет определять типизированные константы.

```
int const daySeconds = 24 * 60 * 60;
int const size = 12;
int const daysInMonths[daySeconds] =
          {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Попытка изменить константные данные приводит к неопределенному поведению

```
int *may = (int*)&daysInMonths[4];
*may = 30;
```

# Указатели и const

Константный указатель и указатель на константу.

```
int a = 10;
const int *firstConstPointer = &a;  // Указатель на константу.
const int *secondConstPointer = &a; // Указатель на константу.
*firstConstPointer = 10;        <- Недопустимо изменения содержания.
secondConstPointer = NULL;          // Допустимо изменение указателя.

int * const pointerToConst = &a;    // Константный указатель.
*pointerToConst = 20;               // Допустимо изменения содержания.
pointerToConst = NULL;          <- Недопустимо изменение указателя.

// Константный указатель на константу.
int const * const constPointerToConst = &a;
*constPointerToConst = 30;      <- Недопустимо изменения содержания.
constPointerToConst = NULL;     <- Недопустимо изменение указателя.
```

# Константные указатели

Можно использовать следующее правильно:
«Слово const делает неизменяемым тип слева от него».

```
int a = 10;
int *pointer = &a;

// Указатель на указатель на константу int.
int const **pointerToPointerToConst = &pointer;

// Указатель на константный указатель на int.
int * const *pointerToConstPointer = &pointer;

// Константный указатель на указатель на int.
int ** const constPointerToPointer = &pointer;
```

# Константные ссылки

Сама по себе является неизменяемой.

```
int a = 10;
int & const reference = a;  <- Ошибка компиляции.
int const & constReference = a;  // Ссылка на константу.
```

Позволяет избежать копирование объектов при передаче в функцию.

```
Point midpoint(Segment const & segment);
```

# Константные методы

Методы классов могут быть объявлены как const.

```
class IntArray {
   int getSize() const;
};
```

Такие методы не могут изменять поля объекта. Неявный указатель this является указателем на const (Type const * this);
У константных объектов (через указатель или ссылку на константу) можно вызывать только константные методы.

```
IntArray const * pointer = foo();
pointer-setSize(10);    <- Ошибка, вызов неконстантного метода.
```

Внутри константных методов можно вызывать только константные методы

# Константные методы

Ключевое слово const является частью сигнатуры метода.

```
int IntArray::getSize() const { return size; }
```

Можно определить две версии одного метода.

```
class IntArray {
   int size;
   int *data;

public:
   int get(int index) const {
      return data[index];
   }

   int& get(int index) {
      return data[index];
   }
```

# Синтаксическая и логическая константность

Синтаксическая константность – константные методы не могут менять поля (обеспечивается компилятором).

Логическая константность – запрещено изменение данных, определяющих состояние объекта в константных методах.

```
class IntArray {
   int size;
   int *data;

public:
   void method() const {
      // Нарушение логической константности.
      data[10] = 1;
   }
};
```

# Ключевое слово mutable

Ключевое слово mutable позволяет определять поля, которые можно изменять внутри константных методов.

```
class IntArray {
   int size;
   int *data;

   mutable int counter;

public:
   int size() const {
      ++counter;
      return size;
   }
};
```

# Отличие структур и классов

Структуры и классы – близкие родственники.

Единственное различие состоит в том, что по умолчанию члены класса являются закрытыми, а члены структуры – открытыми.

В соответствии с формальным синтаксисом C++ объявление структуры создает тип класса.

Структуры сохранены в C++ для совместимостью с C.

# Разбиение программы на файлы

- С небольшими файлами удобнее работать;
- Разбиение на файлы структурирует код;
- Позволяет нескольким программистам разрабатывать приложение одновременно;
- Ускорение повторной компиляции при небольших изменениях в отдельных частях программы.

# Заголовочный файл

- Имеет расширение .hpp;
- Может содержать только объявление;
- Не должен содержать определения;
- Должен иметь механизм защиты от повторного включения.

```
#ifndef SYMBOL
#define SYMBOL

// Набор объявлений

#endif
```

# Файл реализации

- Имеет расширение .cpp;
- Может содержать как определения так и объявления;
- Объявления будут локальны для данного файла;
- Должен содержать директиву включения заголовочного файла;
- Не должен содержать объявлений, дублирующих объявления в соответствующем заголовочном файле.

```
#include "point2d.h"
```

# Конец лекции