Объектно-ориентированное программирование

Представляет собой логичное развитие структурного программирования.

Основная единица — **класс**, сложный тип данных, сочетающий в себе переменные и функции для работы с ними.

Класс обладает свойствами, недоступными обычным переменным, и позволяет автоматизировать многие рутинные процессы.

На уровне классов вводится разделение доступа, обеспечивающее целостность данных и сокрытие реализации.

Класс

Класс — это сложный тип данных, включающий в себя как переменные, так и функции для работы с ними.

Переменные внутри класса носят название «**Поля**».

Функции, объявленные внутри класса, носят название «**Методы**».

Экземпляр класса (переменная данного типа) носит название **объект**. Разные объекты одного класса обладают одинаковыми методами и полями, но разными значениями полей.

```
class Storage
{
    char* array;
    int   size;
}
```

Наследование

Механизм **наследования** позволяет создавать новые классы на основе уже существующих.

Исходный класс называется **родительским классом** или **суперклассом**.

Производный класс также называют **дочерним классом**.

Дочерний класс обладает всеми теми же полями и методами, что и родительский, а также может добавить свои поля и методы, либо переопределить родительские методы, заменив их реализацию.

```
class Keeper : public Storage
{
    char name[128];
    // Поля array и size также доступны в данном классе
}
```

Инкапсуляция

Механизм **инкапсуляции** позволяет скрыть реализацию класса и обеспечить сохранность его внутренней структуры.

На уровне класса определены уровни доступа:

- **public** — все поля и методы на данном уровне доступы всем.
- **protected** — все поля и методы на данном уровне доступны классу и его потомкам.
- **private** — все поля и методы на данном уровне доступны только классу и недоступны даже потомкам.

Все поля класса по умолчанию объявлены как **private**.

В языке C++ структура представляет собой класс, все поля и методы которого по умолчанию объявлены как **public**.

Разделение объявлений и определений методов скрывает реализацию и упрощает читаемость.

Инкапсуляция

```
class Storage
{
public:
  void set_array(
    char* ar,
    int sz
  );
private:
  char* array;
  int   size;
}
```

```
void Storage::set_array(
  char* ar, int sz)
{
  array = new char[sz];
  memcpy(array, ar, sz);
  size = sz;
}

int main()
{
  Storage st;
  st.set_array(
    new char[1024], 1024
  );
  st.size = 128;  // !!!!
}
```

Конструктор и деструктор

Конструктор — это специальный метод, вызываемый явно или неявно при создании класса. Задача конструктора — привести поля класса в их начальное состояние.

Конструктор — это метод без возвращаемого значения, называющийся так же, как и сам класс.

Класс может содержать несколько конструкторов, различающихся аргументами.

Деструктор — это специальный метод, вызываемый неявно при удалении экземпляра класса. Задача деструктора — освободить ресурсы, которые класс мог занять в процессе работы.

Деструктор — это метод без возвращаемого значения и аргументов, называющийся так же, как сам класс, но перед его именем ставится символ ~ (тильда).

В классе всегда только один деструктор.

Конструктор и деструктор

```
class Storage
{
public:
  Storage();
  Storage(char* ar, int sz);
  ~Storage();
  void set_array(char* ar, int sz);

private:
  char* array;
  int   size;
}
```

Конструктор и деструктор

```
Storage::Storage()
{
  array = nullptr;
  size = 0;
}

Storage::Storage(char* ar, int sz)
{
  array = new char[sz];
  memcpy(array, ar, sz);
  size = sz;
}

Storage::~Storage()
{
  delete [] array;
}
```

Конструктор и деструктор

```
#include <string.h>

int main()
{
  Storage st1;                  // Вызов первого конструктора
  char* s = "Hello world\n";
  Storage st2(s, strlen(s)); // Вызов второго конструктора
  return 0;
} // Вызов деструкторов обоих объектов
```

Полиморфизм

Полиморфизм — это свойство производного класса менять поведение методом родительского класса, переопределяя их реализацию.

Также, в C++ полиморфизм — это возможность создавать несколько функций с одинаковым именем, но разными аргументами.

Компилятор автоматически выбирает нужную функцию в зависимости от типа переданного значения.

Полиморфизм

```
int power(int x, int p)
{
  int res = 1;
  for(int a=0; a<p; a++) res *= x;
  return res;
}

double power(double x, int p)
{
  double res = 1;
  for(int a=0; a<p; a++) res *= x;
  return res;
}
```

Полиморфизм

```
int power(int x, int p);

double power(double x, int p);

int main()
{
  power(2, 10);     // Вызов функции power для типа int
  power(2.0, 10);   // Вызов функции power для типа double
}
```

Пространства имен

Пространство имен — это идентификатор, используемый для размещения в нем других идентификаторов.

Пространство имен — аналог папки для файлов.

Внутри одного пространства имен не допускаются одинаковые идентификаторы (кроме одинаковых имен полиморфных функций). Однако можно разместить одинаковые идентификаторы в разных пространствах имен и избежать коллизии.

Каждый класс определяет пространство имен, одноименное с ним. Это позволяет выносить определения методов класса за пределы объявления класса.

Всегда существует безымянное пространство имен.

Пространства имен

Для использования идентификатора из пространства имен std необходимо записать его имя и два двоеточия:

```
std::string s;
```

При необходимости частой использовании пространства имен, можно подключить его целиком:

```
using namespace std;
string s;
```

Безымянное пространство имен не имеет идентификатора и адресуется просто через два двоеточия:

```
::printf("Hello world");
```

При записи без двоеточий используются все доступные пространства имен, в том числе безымянное.

Пространства имен

```
#include <math.h>
namespace simplemath
{
  int power(int x, int p)
  {
      int res = 1;
      for(int a=0; a<p; a++) res *= x;
      return res;
  }
}

int main()
{
  int i = simplemath::power(2, 10);
  double d = ::pow(2, 10);
}
```

Виртуальные методы

Производный класс всегда содержит в себе методы родительского класса.

Прямое объявление метода с тем же именем и аргументами в производном классе создаст копию метода, и производный класс будет содержать оба метода.

Приведение производного класса к типу родительского позволяет вызвать родительский метод.

Виртуальные методы

```
class Base
{
public:
  void name()
  {
    printf("Base");
  }
}
```

```
class Derived: public Base
{
public:
  void name()
  {
    printf("Derived");
  }
}
```

```
Base b;
Derived d;
b.name();   // Base
d.name();   // Derived
```

Виртуальные методы

```
class Base
{
public:
  void name()
  {
    printf("Base");
  }
}
```

```
class Derived: public Base
{
public:
  void name()
  {
    printf("Derived");
  }
}
```

```
Derived d;
Base* b = &d;
d.name();   // Derived
b->name();  // Base
```

Виртуальные методы

Виртуальный метод — это метод, конкретная реализация которого определяется классом, к которому принадлежит объект, то есть типом объекта.

Реализация виртуального метода выбирается во время выполнения программы и не зависит от того, к какому типу приведен объект.

Для объявления виртуального метода используется ключевое слово **virtual**.

Родительский класс, содержащий виртуальные методы, определяет общий интерфейс работы для всего семейства классов, а дочерние классы только меняют конкретную реализаци. Пример: семейство классов ввода-вывода (файл, сеть, последовательный порт).

Виртуальные методы

```
class Base
{
public:
  virtual void name()
  {
    printf("Base");
  }
}
```

```
class Derived: public Base
{
public:
  void name()
  {
    printf("Derived");
  }
}
```

```
Derived d;
Base* b = &d;
d.name();   // Derived
b->name();  // Derived
```

Ссылки

Ссылка — это способ создать новое имя для той же переменной.

Ссылка похожа на указатель, но безопаснее в работе.

Ссылки используются для передачи больших объектов, чтобы избежать большого копирования данных.

После создания ссылка сразу должна быть связана с объектом и не может быть связана с другим объектом в процессе работы.

Для создания ссылок используется символ & (амперсанд) между типом и именем переменной. После создания ссылки сразу следует присвоение для установки соответствия между ссылкой и объектом.

Созданная ссылка ничем не отличается в работе от самого объекта.

Ссылки

```
// Функция принимает ссылку на объект типа Storage
void init_storage(Storage& s, int size)
{
  // Ссылка работает так же, как и обычный объект
  s.set_array(new char[size], size);
}

int main()
{
  Storage ss;
  init_storage(ss);   // Передача объекта по ссылке
                      // Данные не копируются
}
```

Потоки ввода-вывода

Язык C++ позволяет переопределять операторы.

На уровне стандартной библиотеки языка переопределены операторы **<<** и **>>**:

- **<<** - Вывод в поток данных.
- **>>** - Ввод из потока данных.

В заголовочном файле **iostream** объявлены стандартные потоки: поток ввода (**cin**), поток вывода (**cout**) и поток вывода ошибок (**cerr**). Для их использования необходимо пространство имен **std**.

Существуют потоки ввода-вывода для файлов и строк.

Также можно разрабатывать собственные потоки ввода-вывода, либо определять операторы ввода и вывода для новых типов данных.

Потоки ввода-вывода

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Input number: ";
  int i;
  cin >> i;
  cout << "The number entered: " << i << endl;
  return 0;
}
// Input number: 42
// The number entered: 42
```

Динамическая память

В языке C работа с динамической памятью вынесена в отдельный заголовочный файл — **stdlib.h**.

За выделение динамической памяти отвечает функция **malloc** (сокращение от memory allocation). Память выделяется в байтах и никак не инициализируется: в памяти лежат случайные значения. Функция возвращает указатель на нетипизированные данные: **void***.

Программист вынужден пересчитать количество переменных в динамическом массиве в байты, используя оператор sizeof, и привести тип **void*** к типу указателя на нужные ему данные (например, **int***).

Освобождение ранее выделенной памяти делается при помощи функции **free**.

```
int* i = (int*)malloc(1024 * sizeof(int));
free(i);
```

Динамическая память

В языке C++ для выделения динамической памяти используется оператор **new**.

При выделении массива указывается количество элементов, а не количество байт. **new** автоматически вызывает конструктор для каждого создаваемого объекта.

Освобождение ранее выделенной памяти делается при помощи оператора **delete**.

Смешивание вызовов **malloc/free** и **new/delete** не допускается.

```
char* c = new char[1024];

Storage* s = new Storage(c, 1024);
delete s;

// delete [] c;
```

Шаблоны

Шаблон позволяет определить функцию или класс, когда какой либо используемый тип неизвестен заранее.

Разворачивание шаблона происходит на этапе компиляции: компилятор создает отдельную версию функции для каждого варианта ее использования. В силу этого, шаблонную функцию нельзя собрать один раз и положить в библиотеку, и как правило ее код лежит в заголовочном файле.

Для определения шаблона используются угловые скобки.

Типовое применение шаблонов — классы-контейнеры, чья логика не зависит от конкретного хранимого ими типа.

Шаблоны

```
template <typename T>
T power(T x, int p)
{
  T res = 1;
  for(int a=0; a<p; a++) res *= x;
  return res;
}

int main()
{
  double d = power<double>(4, 5); // Явное указание типа
                                  // Замена T -> double

  double d2 = power(d, 2);  // Тип double угадывается
                            // компилятором автоматически
}
```

Стандартная библиотека C++

Стандартная библиотека C++ содержит набор классов для решения рутинных задач. Все эти классы расположены в пространстве имен **std**.

- **string** — используется для хранения 8-битных строк. Автоматически выделяет память по мере необходимости. Заголовочный файл **string**.
- **fstream** — используется для файлового ввода-вывода, аналогично потокам **cin** и **cout**. Заголовочный файл **fstream**.
- Шаблонные классы-контейнеры: vector, list, deque, map, set. Позволяют хранить произвольное количество однотипных элементов. Отличаются внутренней структурой и разной приспособленностью под одинаковые операции. Используют одноименные заголовочные файлы.