Unions

Контейнер - набор из элементов РАЗНОГО типа.
Контейнер в стиле Си - юнион.

Минусы:

- Добавление нового типа может повлиять на максимальнй размер юниона, вызвав бинарную несоместимость.
- В случае C++ внутри юниона будут типы с конструкторами

> `new (&obj) smth` - создать объект динамически по заданному адресу &obj.

> `= default;` не всегда означает пустой конструктор/дструктор. В некоторых экзотических контекстах (в юнионе) он может означать `= delete;`

Другой подход - `void*`

```
struct Node {
    Node *parent_;
    Node_t type_; // type of node
    void *data_; // ptr to derived struct (BinOp)
};

struct BinOp {
    Node basepart_;
    BinOp_t op_;
    Node *lhs_; *rhs_;
}
```

Фактически данная идея отражает механизм C++ наследования

> По указателю на базовый класс можно получить доступ к производному.

Наследование - это отношение "является": *Derived is a Base.*

Принцип подстановки Лисков

> Два признака использования наследования, которые должны выполняться вместе:
>
> - Производный **расширяет** базовый
> - Производный является **частным случаем (is a)** базовым

> Что от чего наследовать - квадрат от прямоугольника, или прямоугольник от квадрата - circle & elipce problem.

```
void foo(const Base &obj);

Derived *dptr = new Derived{};
foo(*dptr);

Base bptr = dptr; // ok
dptr = static_cast< Derived* >(bptr); // ok (иначе срезка за искл. вирт. методов)
```

Проблема срезки

```
// some case
struct A {
    int a_;
    A(int a) : a_(a) {}
};

struct B : public A {
    int b_;
    B(int b) : A(b / 2), b_(b) {}
};

B b1(10); // {5 10}
B b2(8); // {4 8}
A& a_ref = b2;
a_ref = b1; // a_ref.operator=(b1); - оператор не полиморфный (там внутри a_
= right.a_; )!!!!!
```

Еще типичный кейс срезки:

```
struct A {
    virtual void dump() { std::cout << "A" << std::endl; }
};
struct B : public A {
    virtual void dump() { std::cout << "B" << std::endl; }
};

void foo(A &obj) { obj.dump(); }
void boo(A obj) { obj.dump(); }

int main() {
    B obj{};

    foo(obj); // not copy, not cut, prints B
    boo(obj); // copy constructor, cut, prints A
}
```

EBCO (Empty Base Class Optimization)

```
class A{}; // base class without state (but can contain methods)
class B : public A{};

A a{}; // sizeof(a) = 1;
B b{}; // sizeof(b) = 1; NOT 2! It is optimization
```

Optimization case example:

```
template <typename T, typename Deleter = default_delete<T>>
class my_uniq_ptr {
    T *ptr_;
    Deleter del_;
    // ...
};
```

Если `Deleter` есть пустой класс, то засчет выравнивание по максимальному размеру (в данном случае по размеру `ptr_`) получим вдвое больший размер. *Если же мы отнаследуемся от `Deleter`, то мы сэкономим память.*

> Подобная экономия памяти делается в том числе засчет использования лямбда функций. Лямбда функция во многом напоминает класс, нежели функцию.

При EBCO в пустом базовом классе должны отсутствовать виртуальные методы, **включая виртуальный деструктор**, потому что виртуальная таблица, как и состояние класса, **требует памяти**.

Полиморфизм ч/з табл. вирт. функицй

```
constructor() : baseConstructor(), fields_init() {} // end of body - end of VTable construction
```

Полиморфизм ч/з виртуальные методы или наследование - динамический полиморфизм
Полиморфизм ч/з шаблоны - статический полиморфизм

Класс интерфейс в C++ - класс с чисто виртуальными методами и чисто виртуальный деструктор.

> Чистая виртуальность не запрещает писать тела у чисто виртуальных методов. Они могут быть вызваны из наследников. Обычно в большинстве случаев интерфейс имеет вид:

```
class Interface {
    virtual void method() const = 0;
    virtual ~Interface{};
};
```

Конструктор не может быть виртуальным.

Виртуальные методы не распознают перегрузки. `virtual void square(int)` вместе с `void square(long) override` выдаст ошибку. При этом `virtual Base* clone()`, `Derived* clone() override` - работает. **Типы, передаваемые в ф-цию КОВАРИАНТНЫ, а из ф-ции КОВАРИАНТНЫ**

protected destructor

Если мы сделаем `protected destructor` у базового класса, то мы теперь не сможем удалить наследников по указателю на базовый класс. Т.е. проблема частичного удаления пропадает. Однако никто не мешает нам удалить наследников на базовый класс внутри одного из наследников :) .

Виртуальные функции и производительность

- Производительность - плохо

> Для виртуализации нужно 1-3 указателя

Виртуализация готова к использованию после тела конструктора класса.

PVC - pure virtual call

Ситуация, при которой происходит КОСВЕННЫЙ вызов чисто виртуальной функции в конструкторе.
Как следствие, любой вызов функции из конструктора и деструктора - невиртуальный.

```cpp
struct Base {
    Base() { doIt(); } // PVC invocation
    virtual void doIt() = 0;
};

struct Derived : public Base { void doIt() override; };

int main() {
    Derived d;
}
```

Таблица виртуальных функций готова после отработки конструктора. Это означает, что вызов виртуальной, чисто виртуальной функций в конструкторе работает как вызов обычной функции. `override` при этом не работает. Таким образом, лучше вызывать виртуальные функции в конструкторах `final` классов.

Статическое и динамическое связывание (в контексте стандарта языка, а не физического связывания)

Функции связываются статически.
Виртуальные функции связываются динамически.
Аргументы по умолчанию связываются статически.
Т.е. нельзя использовать аргументы по умолчанию в рамках виртуальных методов!

```cpp
struct Base {
    virtual int foo(int a = 14) { return a; }
};

struct Derived : public Base {
    int foo(int a = 42) override { return a; }
};

int main() {
    Base *pb = new Derived{};
    std::cout << pb->foo() << std::endl; // print 14!
}
```

Эта проблема решается идиомой NVI (Non Virtual Interface):
помещаем вызов виртуальной функции в невирутальную функцию с аругментами по умолчанию.
Другой способ: сделать `static_cast<Derived*>(pb)->foo()`.

Полиморфизмы в C++

Перегрузки - статический полиморфизм по любому аргументу
Шаблоны - статический полиморфизм по любому аргументу, метапрограммирование на этапе инстанцирования.
Виртуализация - динамический полиморфизм по первому неявному аргументу this.

Отличие закрытого наследования (part-of) от композиции

При закрытом наследовании:

- Можно переопредлять виртуальные функции из базового класса
- Можно получить доступ к protected полям
- Можно использовать `using` и втаскивать имена из скопа родителя в скоп ребенка.
- Нельзя сделать статик каст от Derivde к Base.

При композиции этого всего делать нельзя. Композиция должна быть выбором по умолчанию.

`using` внутри наследников

```
1  struct Matrix {
2      virtual void pow(double x); // 1
3      virtual void pow(int x); // 2
4  };
5
6  struct SparseMatrix : public Matrix {
7      // using Matrix::pow; // (4)
8      void pow(int x) override; // 3
9  }
10
11 int main() {
12     SparseMatrix d;
13     d.pow(1.5);
14 }
```

В namespace SparseMatrix нету имен функций из Matrix, поэтому они не рассматриваются в качестве кандидатов на вызов. Т.к. существует implicit cast из double в int и нету `using`, то 3 -- единственный кандидат!

С использованием 4 вызовется 1.