

**C++**

# Лекция VI

Classes: Alignment, Padding. Inheritance

# Alignment & Padding

# Alignment

- Выравнивание - положительное число, степень двойки (минимальное значение = 1)
- К объектам предъявляются требования по выравниванию в памяти: адрес объекта должен нацело делиться на выравнивание

```
#include <iostream>
#include <type_traits>

int main() {
    std::cout << alignof(int) << std::endl;
    std::cout << std::alignment_of<int>::value << std::endl;
}
```

# Alignment

- процессор вычитывает данные по размерам, кратным машинным словам (32 or 64 bits)
- рассмотрим два расположения `int32_t` в памяти

memory	
+-----+	<-- 104
4	
+-----+	<-- 103
3	
+-----+	<-- 102
2	
+-----+	<-- 101
1	
+-----+	<-- 100

memory	
4	
+-----+	<-- 104
3	
+-----+	<-- 103
2	
+-----+	<-- 102
1	
+-----+	<-- 101
-	
+-----+	<-- 100

- В клетках указан порядковый номер бита

В каком случае чтение эффективнее? (32 bit system)

# Alignment

- компилятор по умолчанию делает эффективнее
  - выравнивает данные
  - платформо- и компиляторо-зависимо
  - пример: на ARM допустимы только выровненные по 4 int'ы

# Alignment & Padding

```
#include <iostream>

struct T {
    char a;
    int32_t b;
};

int main() {
    std::cout << sizeof(T) << " " << alignof(T);
}
```

8 4

memory:

```
|<----- 8 bytes ----->|
+---+---+---+---+---+---+
| a | padding |           |
+---+---+---+---+---+---+
^               ^
|               |
100             104
```

# Alignment & Padding

```
clang++ -Xclang -fdump-record-layouts
```

```
Dumping AST Record Layout
0 | class T
0 |   int32_t b
4 |   char c
  | [sizeof=8, dsize=5, align=4,
  |   nvsize=5, nvalign=4]
```



# Make alignment stricter

```
struct alignas(8) T {  
    char a;  
    int32_t b;  
};
```

```
struct T {  
    alignas(8) char a;  
    int32_t b;  
};
```

```
alignas(64) char cacheline[64];
```

# #pragma pack

```
#pragma pack(push, 1)
class T { // note: `class` => sizeof != dsize
    int32_t b;
    char c;
};
#pragma pack(pop)
class D {
    int32_t b;
    char c;
};
```

Dumping AST Record Layout

```
0 | class T
0 |   int32_t b
4 |   char c
  | [sizeof=5, dsize=5, align=1,
  |   nvsize=5, nvalign=1]
```

Dumping AST Record Layout

```
0 | class D
0 |   int32_t b
4 |   char c
  | [sizeof=8, dsize=5, align=4,
  |   nvsize=5, nvalign=4]
```

# Object and Value Representations

- Object Representation: `sizeof(T)` последовательных объектов типа `unsigned char`
- Value Representation: биты, хранящие значение объекта
  - NB: `sizeof(struct_type) >= sum(sizeof(member_i_type))`

**placement new**

# example 1

```
#include <iostream>

struct alignas(16) T {
    T(size_t sz) : data_(new char[sz]) { std::cout << __PRETTY_FUNCTION__ << std::endl;
    ~T() { std::cout << __PRETTY_FUNCTION__ << std::endl; delete[] data_; }
    char* data_;
};

void f(bool call_dtor) {
    const size_t N = 100;
    alignas(alignof(T)) unsigned char buffer[N*sizeof(T)];
    T* t = new (buffer) T(42);
    if (call_dtor) {
        t->~T(); // !!! do not forget!
    }
}

int main() {
    f(false); // memory leak
    std::cout << "-----" << std::endl;
    f(true);
}
```

T::T(size\_t)

-----

T::T(size\_t)

T::~~T()

# example 2

```
#include <iostream>
#include <new>

struct alignas(16) T {
    T(size_t sz) : data_(new char[sz]) { std::cout << __PRETTY_FUNCTION__ << std::endl;
    ~T() { std::cout << __PRETTY_FUNCTION__ << std::endl; delete[] data_; }
    char* data_;
};

void f(size_t n, bool call_dtor) {
    // auto* buffer = new unsigned char[n]; // not aligned properly!!!
    auto* buffer = new (std::align_val_t{alignof(T)}) unsigned char[n];
    T* t = new (buffer) T(42);
    if (call_dtor) {
        t->~T();
    }
}

int main() {
    f(100, false); // memory leak here
    std::cout << "-----" << std::endl;
    f(100, true);
}
```

T::T(size\_t)

-----

T::T(size\_t)

T::~~T()

# placement new

Детали:

```
new (placement-params) type [initializer]
```

- Выберите нужный **оператор new**
- Отбросьте первый параметр
- Оставшиеся параметры могут быть использованы в качестве placement-params

# new/delete Expression

- new:
  1. memory allocation
  2. object construction
  3. address returning
- delete
  1. object destruction
  2. memory deallocation



# operators

- new/delete *выражения* используют соответствующие **операторы** для выделения памяти
- операторы можно переопределить
  - глобально
  - для отдельного класса (как static-методы)

# example I

```
void * operator new(size_t size) {
    void* p = malloc(size);
    std::cout << "Allocate " << size << " bytes at addr=" << p << std::endl;
    return p;
}

void operator delete(void* p) noexcept {
    std::cout << "Deallocate at addr: " << p << std::endl;
    free(p);
}

struct S {
    S() { std::cout << __func__ << " called" << std::endl;}
    ~S() { std::cout << __func__ << " called" << std::endl; }
};

int main() {
    std::cout << "---- [NEW] ---- " << std::endl;
    S* s = new S;

    std::cout << "---- [DELETE] ---- " << std::endl;
    delete s;
}
```

```
---- [NEW] ----
Allocate 1 bytes at addr=0x55e0f08e4ec0
S called
---- [DELETE] ----
~S called
Deallocate at addr: 0x55e0f08e4ec0
```

# example II

```
struct S {
    S() { std::cout << __func__ << " called" << std::endl; }
    ~S() { std::cout << __func__ << " called" << std::endl; }

    void * operator new(size_t size) {
        void* p = ::operator new(size);
        std::cout << "Allocate " << size << " bytes at addr=" << p << std::endl;
        return p;
    }

    void operator delete(void* p) noexcept {
        std::cout << "Deallocate at addr: " << p << std::endl;
        ::operator delete(p);
    }
};

int main() {
    std::cout << "---- [NEW] ---- " << std::endl;
    S* s = new S;

    std::cout << "---- [DELETE] ---- " << std::endl;
    delete s;
}
```

```
---- [NEW] ----
Allocate 1 bytes at addr=0x55b156a20ec0
S called
---- [DELETE] ----
~S called
Deallocate at addr: 0x55b156a20ec0
```

# Наследование

- отношение is-a между классами:

child is-a parent

Dog is-a(n) Animal

- переиспользование кода:
  - создание классов на основе имеющихся
  - если код работает с указателем\ссылкой на базовый класс, то такой код можно использовать для наследников

# Syntax

```
class|struct derived-class-name:  
    { access-specifier [virtual] base-class-name, ... }  
{ member-specification }
```

access-specifier – public, protected, private

влияет на доступ к открытым членам класса base-class-name в наследнике

# Example

```
struct GameObject{ Point position; };  
struct Prize : GameObject { // public - by default for structs  
    int value;  
};
```

Object Representation:

```
\  
GameObject:  
+-----+  
| position |  
+-----+  
Prize:  
+-----+-----+  
| position | value |  
+-----+-----+  
^         ^  
|GameObject|
```

# Преобразования Base $\leftarrow$ Derived

Определены автоматически(неявно):

```
Prize p{ Point{...}, 100 };  
GameObject &go = p;  
GameObject *goPtr = &p;
```

- Base&  $\leftarrow$  Derived& и Base\*  $\leftarrow$  Derived\* автоматически

---

Если родитель является копируемым типом, тогда он копируем от объекта-наследника:

```
Prize p{ Point{42, 24}, 100 };  
GameObject go = p;    // GameObject(const GameObject&);  
                     // GameObject{Point{42, 24}}
```

- срезка (создание копии): поля только базового класса

# Особенности

- Базовый класс должен быть определен до наследования
- Из наследника нет доступа к private полям базового класса, есть к public и protected

inher-type	access_type		
	public-member	protected-member	private-member
public	public	protected	no-access
protected	protected	protected	no-access
private	private	private	no-access

```
struct A {  
    access_type: // public-members, private-member, protected-member  
    T member;  
}  
struct B : inher-type A {  
    // to determine access to A::member see the table above^  
};
```



# Особенности

exposing protected members

```
class A {  
protected:  
    int a;  
    int m() { return 1; }  
};  
  
class B : public A { // protected,private тоже допустимы  
public:  
    using A::m;  
    using A::a;  
};  
  
int main() {  
    B b1;  
    b1.a = 32;  
    b1.m();  
}
```

# Constructors

- конструкторы не наследуются (by default)
- сконструировать Base-часть до Derived **необходимо**
  - явно или через конструктор по-умолчанию
  - до выполнения списка инициализации полей Derived
- порядок конструирования: в порядке объявления наследования
  - вызовы деструкторов — в обратном порядке

# Constructor inheritance

```
#define LOG_NAME() std::cout << __PRETTY_FUNCTION__ << std::endl
class A {
public:
    A(int) { LOG_NAME(); }
    A(int, int) { LOG_NAME(); }
    A(int, int, int) { LOG_NAME(); }
    A(int, int, int, int) { LOG_NAME(); }
};

class B : public A {
};

int main() {
    B b1(1);
    B b2(1, 2);
    B b3(1, 2, 3);
    B b4(1, 2, 3, 4);
}
```

# Constructor inheritance

```
#define LOG_NAME() std::cout << __PRETTY_FUNCTION__ << std::endl
class A {
public:
    A(int) { LOG_NAME(); }
    A(int, int) { LOG_NAME(); }
    A(int, int, int) { LOG_NAME(); }
    A(int, int, int, int) { LOG_NAME(); }
};

class B : public A {
    using A::A;
};

int main() {
    B b1(1);
    B b2(1, 2);
    B b3(1, 2, 3);
    B b4(1, 2, 3, 4);
}
```

# Constructors

```
struct GameObject {  
    // no default constructor  
    GameObject(Point position) : position{position} {}  
  
    Point position;  
};  
  
struct Prize: GameObject {  
    Prize(Point pos, int val)  
        : GameObject{pos}  
        , val{val}  
    {}  
  
    Prize(Point p)  
        : position(p) // error! not in init-list  
        // error! no default ctor for base GameObject  
    {}  
    // ...  
};
```

# Methods overriding

```
struct GameObject {  
    /* ... */  
    void CalcShift() { /* ... */ }  
};  
  
struct RoadSign: GameObject {  
    /* ... */  
    void CalcShift() {  
        if (HitByCar()) { InitShifting(); }  
        GameObject::CalcShift(); // явный вызов метода базы  
    }  
};  
  
int main() {  
    GameObject go; go.CalcShift();  
    RoadSign rs; rs.CalcShift();  
}
```

- методы совпадают по сигнатуре

# Virtual methods motivation

```
struct GameObject {  
    void CalcShift() { /* ... */ }  
};  
  
struct RoadSign : GameObject {  
    void CalcShift() { /* ... */ }  
};  
  
std::vector<GameObject *go> objects;  
  
int main() {  
    GameObject *go = new RoadSign{...};  
    objects.push_back(go);  
    objects[0]->CalcShift();  
}
```

# Virtual methods syntax

```
virtual member-function [override] [final] [= 0;]
```

`override` – компилятор проверит, что функция с такой сигнатурой есть в предке

`final` – запрет переопределения в потомках

`= 0;` – pure virtual function (class -> abstract class, нельзя создавать объекты)



# Virtual methods example

```
struct GameObject {  
    virtual void CalcShift() {  
        std::cout << __PRETTY_FUNCTION__ << " called" << std::endl;  
    }  
};  
  
struct RoadSign: GameObject {  
    virtual void CalcShift() override {  
        std::cout << __PRETTY_FUNCTION__ << " called" << std::endl;  
        GameObject::CalcShift();  
    }  
};  
  
int main() {  
    GameObject* go = new RoadSign;  
    go->CalcShift();  
    delete go;  
}
```

```
virtual void RoadSign::CalcShift() called  
virtual void GameObject::CalcShift() called
```

# Abstract classes example

```
struct VisibleObject {  
    virtual void Render() = 0;  
}  
  
// допустима своя реализация вне класса  
void VisibleObject::Render() { /* ... */ }  
  
struct RoadSign: VisibleObject {  
    virtual void Render() final { // метод *надо* переопределить  
        /* ... */  
  
        // реализацию можно заиспользовать явно  
        VisibleObject::Render();  
    }  
};
```

# Virtual methods

- реализация — vtable
  - таблица виртуальных функций (в начале класса)
  - создание объекта в т.ч. подставляет адрес на правильный vtable
- **важен** виртуальный деструктор при наследовании
- виртуальные методы **не стоит** использовать в конструкторах и деструкторах
- конструктор не может быть виртуальным

# Destructor

```
struct S {  
    ~S() { std::cout << __PRETTY_FUNCTION__ << std::endl; }  
};  
  
struct T : S {  
    ~T() { std::cout << __PRETTY_FUNCTION__ << std::endl; }  
};  
  
int main() {  
    S* s = new T;  
    delete s;  
}
```

S::~~S()

# Virtual Destructor

```
struct S {  
    virtual ~S() { std::cout << __PRETTY_FUNCTION__ << std::endl; }  
};  
  
struct T : S {  
    virtual ~T() { std::cout << __PRETTY_FUNCTION__ << std::endl; }  
};  
  
int main() {  
    S* s = new T;  
    delete s;  
}
```

```
virtual T::~~T()  
virtual S::~~S()
```

# Object Representation

```
struct GameObject {
    GameObject(Point position) : position{position} {}

    virtual void CalcShift() { }

    Point position;
};

struct Prize: GameObject {
    Prize(Point pos, int val) : GameObject{pos}, val{val}
    {}
    virtual void CalcShift() override {
        GameObject::CalcShift();
    }
    int val;
};

int main() {
    Prize p{ Point{2.0, 2.2}, 3};
}
```

# Dumping record layouts

```
-Xclang -fdump-record-layouts
```

```
Dumping AST Record Layout
      0 | struct GameObject
      0 |   (GameObject vtable pointer)
      8 |   struct Point position
      8 |     double x
     16 |     double y
      | [sizeof=24, dsize=24, align=8,
      |   nvsize=24, nvalign=8]
```

```
Dumping AST Record Layout
      0 | struct Prize
      0 |   struct GameObject (primary base)
      0 |     (GameObject vtable pointer)
      8 |     struct Point position
      8 |       double x
     16 |       double y
     24 |   int val
      | [sizeof=32, dsize=28, align=8,
      |   nvsize=28, nvalign=8]
```

# vtables layouts

```
-Xclang -fdump-vtable-layouts
```

```
Vtable for 'Prize' (3 entries).
```

```
0 | offset_to_top (0)
1 | Prize RTTI
   -- (GameObject, 0) vtable address --
   -- (Prize, 0) vtable address --
2 | void Prize::CalcShift()
```

```
VTable indices for 'Prize' (1 entries).
```

```
0 | void Prize::CalcShift()
```

```
Vtable for 'GameObject' (3 entries).
```

```
0 | offset_to_top (0)
1 | GameObject RTTI
   -- (GameObject, 0) vtable address --
2 | void GameObject::CalcShift()
```

```
VTable indices for 'GameObject' (1 entries).
```

```
0 | void GameObject::CalcShift()
```



