

Лекция 4. Встроенные типы, инструкции и операторы.

Встроенные типы

Группа	Тип	Литералы
Логический	bool	true, false
Символьный	char, wchar_t	'a', '\n', '\t', L'xy'
Целый	int	0, 5, 0xef, 045
	short [int]	32767
	long [int]	56L, -1234567L
	unsigned [int]	765u
Вещественный	long double	6.626e-34L
	double	2.71, .56, 2., 6e24
	float	3.1415f

Размеры типов

1	1	== sizeof(char)	<= sizeof(short)	<= sizeof(int)	<= sizeof(long)
2	1	<= sizeof(bool)	<= sizeof(long)		
3	sizeof(char)	<= sizeof(wchar_t)	<= sizeof(long)		
4	sizeof(float)	<= sizeof(double)	<= sizeof(long double)		
5	sizeof(T)	== sizeof(signed T)	== sizeof(unsigned T)		

- Ограничения типа:
`std::numeric_limits<type>`
- Не полагайтесь на размер, для этого есть
`uint16_t`, `int64_t` и т.д.

Явное приведение типов

- Базовые типы неявно приводятся друг у другу. Многие с потерей значимости (warning).

```
1. float*      pfv;  
2. const int*  piv;  
3. short      sv;  
4.  
5. int* a = (int*)pfv; // 1. C-style  
6. char b(sv);         // 2. constructor  
7.  
8. double pi  = 3.1415;  
9. int      ipi = static_cast<int>(pi);  
10. //---  
11. int* c = reinterpret_cast<int*>(pfv);  
12. //---  
13. int* d = const_cast<int*>(piv);  
14. //---  
15. derived* d = dynamic_cast<derived*>(base);
```

Тип void

```
1. void *p = 0, *q = 0;
2. bool eq = (p == q);
3. void do_something(int value);
4. int main (void); // C-style
5. void nothing;      // wrong!
```

- Указатель на void:

```
1. void send(const void* data, size_t size);
2. //...
3. X* ptr = ...;
4. send(ptr, sizeof(X));
5. //-----
6. void receive_X(const void* data, size_t size)
7. {
8.     Assert(size == sizeof(X));
9.     X const* ptr = static_cast<data>;
10. }
```

null terminated strings

```
1. const char str [] = "Hello";           // native type
2. char      can_edit[] = "can change this string";
3. char*     c_style   = "old style string"; // don't change it
4.
5. const char m1 [] = "this is"
6.                  "multiline\
7.                  string";
```

- Одинаковые литералы могут размещаться в одном месте
- Спец. символы: `'\t'` `'\r'` `'\n'` `'\a'`

null terminated string	std::string
<code>char* a = "...", *b = "...";</code>	<code>string a = "...", b = "...";</code>
<code>strcmp(a, b) == 0;</code>	<code>a == b;</code>
<code>strcpy(a,b);</code>	<code>a = b;</code>

Перечисления `enum`

```
1. enum msg_type
2. {
3.     mt_setup,
4.     mt_request = 0x10,
5.     mt_response
6. };
```

- Нет неявного приведения из целого типа
- Пользовательский тип - можно, например, переопределить операторы

```
1. int process_msg(msg_type type, const void* data)
2. {
3.     switch(type)
4.     {
5.         case mi_setup    : return process_setup    (*static_cast<setup    *>(data));
6.         case mi_request  : return process_request  (*static_cast<request  *>(data));
7.         case mi_response : return process_response(*static_cast<response*>(data));
8.         default: throw std::runtime_error("unknown message");
9.     }
10. } // hidden error! where?
```

Синоним имени `typedef`

1.	<code>typedef</code>
2.	<code>std::map<std::string, student></code>
3.	<code>students_t;</code>

- Задаёт синоним имени, не задаёт новый тип
- Делает код короче – проще читать
- Платформонезависимые типы
- «Протаскивает» тип через шаблон
- Имеет внутреннюю компоновку

И еще раз про объявления

Необязательный спецификатор (e.g. extern, virtual)	Базовый тип	Объявляющая часть	Необязательный инициализатор
---	const char	array[]	= "Hello, World!"

- *Идентификаторы*: буквы, цифры, _ (начало не с цифры). Длина не ограничена стандартом, но часто реализацией
- Объявление через запятую – остается только базовый тип (не делайте так!):

1.	<code>int* x, y = 1, *const z = 0;</code>
2.	<code>// int* x; int y = 1; int *const z = 0;</code>

Области действия и видимости

```
1.  int uno; // = 0
2.
3.  void foobar()
4.  {
5.      int due; // = 'trash'
6.      int& quatro = due;
7.
8.      int uno = 1;
9.      ::uno = 2;
10.     // uno != ::uno
11.
12.     if (true)
13.     {
14.         double due = 8.31;
15.         T tre = func(&tre);
16.         quatro = 2; // due <line 5> == 2
17.     }
18. }
```

Структуры `struct`

```
1. struct student
2. {
3.     string name;
4.     double av_score;
5. };
6.
7. student peter = {"Peter", 4.68};
8. student* paul = new student;
9.
10. paul->name      = "Paul";
11. paul->av_score = 3.96;
12.
13. struct size_test
14. {
15.     short mem1;
16.     bool  mem2;
17.     int   mem4;
18. };
19.
20. sizeof(size_test) == 8; // not 7, at least on my computer
```

Структуры `struct`, часть 2

```
1. struct list;
2. struct node
3. {
4.     list* owner; // correct
5.     node* next ; // correct
6.     T      data ;
7. };
8.
9. // declarations
10. extern list my_list;
11. list make_list();
12. void sort_list(list&);
13.
14. // definitions
15. list wrong;
16. list* ok;
17. make_list(); // wrong, need size
18. sort_list(wrong); // hmm, wrong?
19.
20. // equivalence
21. // Эти структуры задают разные типы
22. struct my_class {int x; int y;};
23. struct your_class{int x; int y;};
```

Операторы

Семантика	Синтаксис
Область видимости	<code>[namespace class]::name</code>
Выбор члена	<code>object.member</code> <code>ptr->member</code>
Доступ по индексу	<code>pointer[]</code>
Вызов функции	<code>expr(expr-list)</code>
Постфиксный инкремент	<code>lvalue++</code>
Идентификатор типа	<code>typeid({expr type})</code>
Преобразование типов	<code>static_cast<type>(expr)</code>
Взятие размера	<code>sizeof({expr type})</code>
Унарные префиксные операторы	<code>~lvalue</code>
Разыменование	<code>*expr</code>
Выделение/освобождение памяти	<code>new type (expr-list)</code>

Операторы, часть 2

Семантика	Синтаксис
Бинарные арифметические операторы	<code>expr * expr</code>
Сдвиги	<code>expr << expr</code>
Бинарные операторы сравнения	<code>expr < expr</code>
Бинарные логические операторы	<code>expr expr</code>
Условное выражение	<code>expr ? expr : expr</code>
Присваивания	<code>expr = expr</code> <code>expr <=< expr</code> <code>expr += expr</code>
Генерация исключения	<code>throw expr</code>
Запятая	<code>expr, expr</code>

Выражения

- Результат:
 - Расширение к большему (например, *)
 - Логическое выражение `bool`
 - Там, где можно, `lvalue` (`a=b=c`)
- Порядок вычисления в общем случае не определен

1.	<code>f(a) + f(b)</code>
2.	<code>// what would you get here?</code>
3.	<code>if (var & mask == 0){...}</code>
4.	<code>if (0<=x<=99) {...}</code>

- Исключение: ленивое вычисление `&&` и `||` на встроенных типах

Инкремент/Декремент

- Префиксный: изменил, вернул новое значение
- Постфиксный: изменил, вернул старое значение

```
1. int a = ++x;  
2. int b = x += 1;  
3.  
4. int c = x++;  
5. int t;  
6. int d = (t = x, x += 1, t);  
7.  
8. // prefer prefix ++  
9. for (auto it = cont.begin(); it != cont.end(); ++it);
```


Инструкция (statement) выбора

```
1.  if (ptr p = get_pointer(...))
2.      if (connected(p))
3.          disconnect(p);
4.      else
5.          connect    (p);
```

- switch
 - окончание ветки: break, return, throw, exit(0)
 - для определения переменных потребуются скобки {}

```
1.  switch(id)
2.  {
3.      btn_yes:
4.          apply_changes();
5.      btn_no :
6.          close_doc    ();
7.          break;
8.      btn_cancel:
9.          continue_editing();
10.         break;
11.     default:
12.         Log("Unexpected btn " << id);
13. }
```

Циклы

```
1. for (init; condition; modification) {}  
2. while(condition) {}  
3. do{} while(condition)  
4.  
5. // examples  
6. for (size_t i = 0, size = strlen(src); i < size; ++i)  
7.     dest[i] = src[i];  
8.  
9. while(*dest++ = *src++); // don't do this  
10.
```

- Управление циклом:
 - досрочный выход: `break`, `return`, `throw`, `exit(0)`
 - продолжение: `continue`

Комментарии

1.	<code>// one line</code>
2.	<code>/* multi</code>
3.	<code>line */</code>

- Лучше вообще без комментариев, чем нерелевантные комментарии
- Если можно выразить кодом, не пишите комментарии
- Сложный алгоритм – дай ссылку на статью
- Желательно написать:
 - сложный код (например, оптимизация)
 - а-ля разметка (окончание namespace)
 - в заголовке файла (зачем он, copyright)

Вопросы?