Object-Oriented Programming

Naming

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Importance of Naming

- Critical issue for software engineering
- Software Engineers name things constantly
- Naming impacts the readability and comprehension of software
- Good names reduce the cost of development
- Careful selection of names can convey the high-level meaning of a task to the developer

Naming

If you have a good name for a method, you do not need to look at the body - Fowler et al.

Naming

If you have a good name for a method, you do not need to look at the body - Fowler et al.

There are only two hard things in Computer Science: cache invalidation and naming things - Phil Karlton

Naming

If you have a good name for a method, you do not need to look at the body - Fowler et al.

There are only two hard things in Computer Science: cache invalidation and naming things - Phil Karlton

There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors Leon Bambrick

What is a name?

```
          ('a'..'z' | 'A'..'Z' | '_' )
('0'..'9' | 'a'..'z' | 'A'..'Z' | '_' )*
```

- Technical term: *identifier*
- Lowercase letters: 'a'..'z'
- Uppercase letters: 'A'..'Z'
- Digits: '0'..'9'
- Underscore: '_'
- First character: letter or underscore

Only ASCII? MultiLanguage

```
// output "hello" in Russian
void привет() {
    std::cout << __FUNCTION__ << '\n';
}

// output "world" in Greek
void κόσμος() {
    std::cout << __FUNCTION__ << '\n';
}

// output "hello" in Japanese
void こんにちは() {
    std::cout << __FUNCTION__ << '\n';
}

int main() {

    привет();
    κόσμος();
    こんにちは();

    return 0;
}
```

Only ASCII?

```
var data = ["🐶","🐱","🦊", "🐷","🦕","🐬","🦡"]

var 🐶 = 0
var 🐱 = data.count
while 🐶 < 🐱 - 1 {
    var 🦊 = 🐶 + 1
    while 🦊 <  🐱 {
        if data[🐶] > data[🦊] {
            data.swapAt(🐶, 🦊)
        }
        🦊 = 🦊 + 1
    }
    🐶 = 🐶 + 1
}

print(data)
```

Non-ASCII Names

```
          ('a'..'z' | 'A'..'Z' | '_' |
           '\200'..'\377' )
('0'..'9' | 'a'..'z' | 'A'..'Z' | '_' |
           '\200'..'\377' )*
```

- C++ standards only allow certain Unicode non-ASCII letters
- Other languages allow Unicode characters, e.g., Swift, even allowing emojis
- Some compilers support extensions of this, e.g., clang
- As used in srcML TextLexer.g. Note: C++ strings can look like names at the start
- Some parts of your toolchain may handle only ASCII, so stick to ASCII unless needed
- How
in a t
sure

Exceptions to the Rules

- Reserved Words: C++ Reserved Words

- *contextual keywords* - identifiers with special meaning in a specific context

- Avoid starting with an underscore '_' as it is reserved for system use

Exception: C++ Keywords

alignas	alignof	and	and_eq	asm	atomic_cancel
atomic_commit	atomic_noexcept	auto	bitand	bitor	bool
break	case	catch	char	char8_t	char16_t
char32_t	class	compl	concept	const	consteval
constexpr	constinit	const_cast	continue	co_await	co_return
co_yield	decltype	default	delete	do	double
dynamic_cast	else	enum	explicit	export	extern
false	float	for	friend	goto	if
inline	int	long	mutable	namespace	new
noexcept	not	not_eq	nullptr	operator	or
or_eq	private	protected	public	reflexpr	register
reinterpret_cast	requires	return	short	signed	sizeof
static	static_assert	static_cast	struct	switch	synchronized
template	this	thread_local	throw	true	try
typedef	typeid	typename	union	unsigned	using
virtual	void	volatile	wchar_t	while	xor
xor_eq					

Exception: Contextual Keywords

Language				
C++	final	override	transaction_safe	transaction_safe_dynamic
C++	import	module		
C#	add	and	alias	ascending
C#	args	async	await	by
C#	descending	dynamic	equals	file
C#	from	get	global	group
C#	init	into	join	let
C#	managed	nameof	nint	not
C#	notnull	nuint	on	or
C#	orderby	partial	record	remove
C#	required	scoped	select	set
C#	unmanaged	value	var	when
C#	where	with	yield	

Exception: System Use

`_Compare`	`__enum_type_info`	`__msg_ctime_high`	`__lcomp`
`__nmemb`	`__min_element_switch`	`__copy_range1`	`_SC_ULONG_MAX`
`__nent`	`__advances`	`__pattern_includes`	`__brick_min_element`
`__blkcnt64_t`	`__outbuf`	`__parallel_for_body`	`__unused2`
`__align`	`__sizep`	`__s`	`_SC_BC_BASE_MAX`
`_CS_POSIX_V6_LPBIG_OFFBIG`	`_rt`	`_SC_REGEXP`	`_CASable_bits`
`_CS_LFS64_LIBS`	`__attrp`	`_IO_buf_base`	`__replace_if_switch`
`_M_z_beg`	`__combine`	`_SC_FILE_ATTRIBUTES`	`__ptr`
`__SYSMACROS_DEFINE_MINOR`	`__target_bin`	`__iov`	`_M_first_insert`
`__gthread_key_create`	`__ISwalpha`	`__cmd`	`__tgid`
`__logp`	`__rseq_offset`	`__cc_range`	`_M_ik`
`_SC_FD_MGMT`	`__collector_t`	`_SC_2_C_DEV`	`__h`
`__x`	`__seqs_end`	`_SC_PII_INTERNET_STREAM`	`__m2`
`_M_ys`	`__size_part`	`__timebuf`	`__argv`

Lots to Name

- variables
- fields (data members)
- parameters
- types
- functions
- methods
- classes

General Rules

- Names should not include *types* or any symbol for the types. I.e., the Hungarian Notation is not a good style
- Variables, parameters, types, classes, fields, classes, files, are *entities*, i.e., *things* and therefore use *nouns*
- Functions and methods are *actions* and therefore use *verbs*
- For most functions/methods, the action is in the *imperative form*, e.g., use `parse()` instead of `parsing()` or `parsed()`
- Will focus on function/method names for now

Operator Overloading: Built-In Types

```
void f() {
    int n1 = 4;
    int n2 = 5;
    int totalN = n1 + n2;

    double x1 = 4.0;
    double x2 = 5.0;
    int totalX = x1 + x2;
}
```

```
__Z1fv:                                 ; @_Z1fv
        .cfi_startproc
; %bb.0:
        sub     sp, sp, #48
        .cfi_def_cfa_offset 48
        mov     w8, #4
        str     w8, [sp, #44]
        mov     w8, #5
        str     w8, [sp, #40]
        ldr     w8, [sp, #44]
        ldr     w9, [sp, #40]
        add     w8, w8, w9
        str     w8, [sp, #36]
        fmov    d0, #4.00000000
        str     d0, [sp, #24]
        fmov    d0, #5.00000000
        str     d0, [sp, #16]
        ldr     d0, [sp, #24]
        ldr     d1, [sp, #16]
        fadd    d0, d0, d1
        fcvtzs  w8, d0
        str     w8, [sp, #12]
        add     sp, sp, #48
        ret
        .cfi_endproc
                                        ; -- End function
```

Function Overloading

```
int sum(int n1, int n2);

double sum(double x, double y);

long sum(long l1, long l2);

int sum(std::vector<int>::const_iterator begin,
        std::vector<int>::const_iterator end);
```

- C++, and most modern programming languages, allow *function overloading*, while C does not

- Can use the same name for multiple functions as long as the parameter list is different (number, types)

- Uniqueness is **not** influenced by the return type

- Name is based on: function name, cv-qualifiers, parameter types

g++ -S struct.cpp

```cpp
struct S {
    int field1;
    int field2;
};

void f() {
    S s;
    s.field1 = 0;
    s.field2 = 1;
}

int main() {

    f();

    return 0;
}
```

```asm
    .globl  __Z1fv ## -- Begin function _Z1fv
    # ...
__Z1fv:             ## @_Z1fv
    # ...
    movq    %rsp, %rbp
    # ...
    movl    $0, -8(%rbp)
    movl    $1, -4(%rbp)
    popq    %rbp
    retq
    .cfi_endproc
                    ## -- End function
    .globl  _main   ## -- Begin function main
_main:              ## @main
    # ...
    callq   __Z1fv
    # ...
    retq
```

Executable Environment

```
    .globl  __Z1fv ## -- Begin function _Z1fv
    # ...
__Z1fv:             ## @_Z1fv
    # ...
    movq    %rsp, %rbp
    # ...
    movl    $0, -8(%rbp)
    movl    $1, -4(%rbp)
    popq    %rbp
    retq
    .cfi_endproc
                    ## -- End function
    .globl  _main   ## -- Begin function main
_main:              ## @main
    # ...
    callq   __Z1fv
    # ...
    retq
```

- C language has functions (with no overloading) and `structs`

- CPU supports assembly labels that refer to a memory address

- CPU supports instructions to call a function located at a memory location (`callq`, `call`, `call1`, `bl`) and return from a function (`retq`, `ret`, `retl`)

Mangling

- *mangling*, or *name mangling*

- method/function names translated into Assembly labels

- There is **no** standard name mangling for C++

- Many compilers follow the Titanium C++ ABI (Application Binary Interface)

C++ Function	Assembly Name
`action()`	_Z6actionv
`action(int)`	_Z6actioni
`action(double)`	_Z6actiond
`action(char)`	_Z6actionc
`action(double&)`	_Z6actionRd
`action(double const&)`	_Z6actionRKd
`action(double*)`	_Z6actionPd
`action(double const*)`	_Z6actionPKd
`action(std::vector<int>&)`	_Z6actionRNSt3__16vectorIiNS_9allocatorIiEEEE
`action(const std::vector<int>&)`	_Z6actionRKNSt3__16vectorIiNS_9allocatorIiEEEE

C++ Function Naming Tools

Description	Command
Compile to Assembly	`g++ -S overloading.cpp`
Compile to x86 Assembly on Apple Silicon Mac	`g++ -target x86_64 -S overloading.cpp`
List symbols in object file	`nm overloading`
Demangle a symbol	`c++filt __Z6actioni`
Demangle all symbols in object file	`nm --demangle overloading`