MODERN C++ DESIGN PATTERNS

Complex Declarations by Prasanna Ghali

Complex Declarations

- Unscrambling complex declarations
- Simplifying complex declarations in C/C++ code

Complex Declaration: Example

- Given following declaration, what is type of foo?

```
char *(*foo(char *, int))[5];
```

How a Declaration is Formed

- Declaration's come in two parts:
  - *base type* consists of type-specifier, storage-class specifier, and type-qualifier
  - *declarator* containing the identifier, or name being declared with characters *, [ ], and ( ) and possibly type-qualifiers

Declarator

How many	Name in C	How it looks in C

zero or more	pointers	one of the following alternatives:
		`* const volatile`
		`* volatile`
		`*`
		`* const`
		`* volatile const`

exactly one	direct declarator	*identifer* or *identifer*[*optional_size*] ... or *identifer* (*args*...) or (*declarator*)
zero or one	initializer	= *initial_value*

Declarations

How many	Name in C	How it looks in C

at least one type-specifier
 	
(not all combinations are valid)
	type-specifier	`void char short int`
		`long signed unsigned`
		`float double`
		`struct_specifier`
		`enum_specifier`
		`union_specifier`

	storage-class	`extern static register`
		`auto typedef`

	type-qualifier	`const volatile`
exactly one	declarator	*`see previous definition`*
zero or more	more declarators	`, declarator`
one	semi-colon	`;`

Restrictions on Legal Declarations

- Function can’t return a function, so `peach()()` never arises
- Function can’t return an array, so `apple()[]` never arises
- Array can’t hold a function, so `orange[]()` never arises

What's Allowed

- You can write any of these declarations:
  - `int (* grape())();`
  - `int (* pear())[];`
  - `int (* mango[])();`
  - `int kiwi[3][4][5];`

Precedence Rule

A. Declarations are read by *starting with the name and then reading in precedence order*

B. Precedence, from high to low, is:

   1. Parentheses grouping together parts of a declaration
   2. Postfix operators:

      Parentheses ( ) indicating a function, and

      Square brackets [ ] indicating an array
   3. Prefix operator * denoting "pointer to"

C. If `const` and/or `volatile` keyword is next to type specifier [`int`, `long`, etc.] it applies to type specifier. Otherwise, `const` and/or `volatile` keyword applies to pointer asterisk on its immediate left

Precedence Rule: Examples

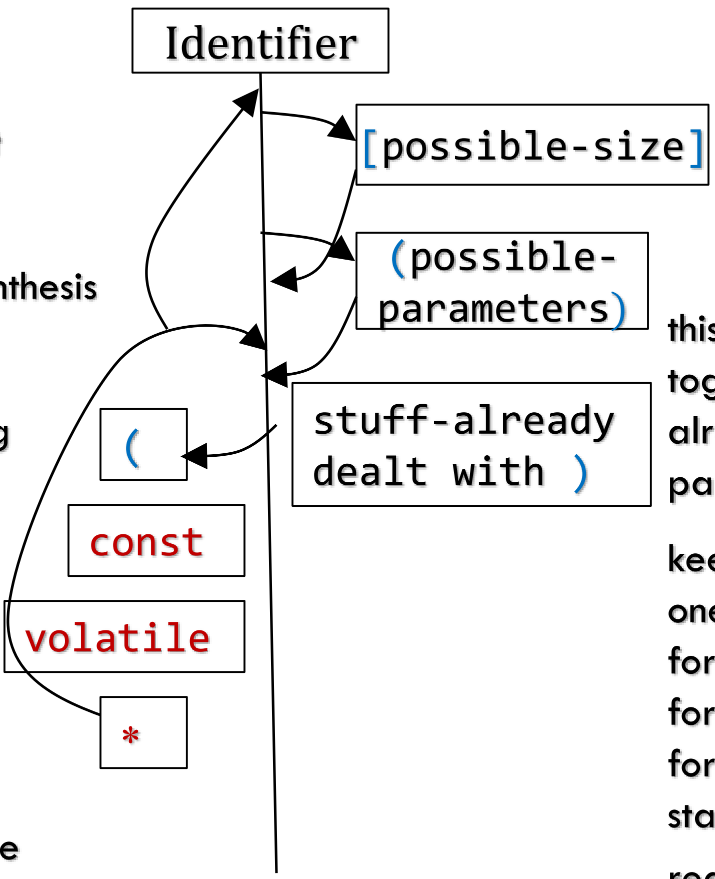
- `char * const * (*next)();`
- `char * (*c[10]) (int **p);`

Unscrambling Declarations by Diagram (1/2)

- Declarations in C are read boustrophedonically, i.e. alternating right-to-left with left-to-right
- Start at first identifier you find when reading from the left
- When a token in declaration is matched against diagram, erase it from further consideration
- At each point, look first to token to right, then to the left
- When everything has been erased, the job is done

Unscrambling Declarations by Diagram (2/2)

Token to match



Step Number	How to read
1. Go to leftmost identifier	say *"identifier is"*
2. Look at next token to **right** if it is a square bracket.	for each pair, say **"array of"**
3. or if it is an opening parenthesis	Read up to closing parentheses say **"function returning"**
4. If token to **left** is a opening parenthesis.	this is opening parentheses, grouping together part of declaration we've already dealt with. Read up to balancing parenthesis. start again at step 2
5. If token to **left** is any of `const`, `volatile` or `*`	keep reading tokens to left until it is not one of these three: for `const` say "read-only" for `volatile` say "volatile" for `*` say "pointer to" start again at step 4
6. Tokens that remain form the basic type of declaration	read off tokens that remain, e.g. `static unsigned int`

Simplifying Complex Declarations

- Use `typedef` storage specifier in both C and C++ code to write type aliases
- Use trailing return type syntax since C++11 to simplify function declarations
- Use keyword `using` since C++11 to write type aliases

`typedef`: Introduction

- `typedef` introduces new name for *existing* type
  - Doesn’t define variable
  - Doesn’t declare new types
  - Just new name for existing type!!!
- Only purpose is to allow you to replace complex type name with simpler mnemonic

typedef: Syntax

any standard or derived type

```
typedef  existing-type IDENTIFIER;
```

keyword

mnemonic traditionally in uppercase

typedef: Examples (1/4)

```
typedef int64_t BIGINT;
BIGINT big_val = 0xabcdef1298765432;
```

```
typedef uint32_t BOOLEAN;
BOOLEAN const True  = 1;
BOOLEAN const False = 0;
BOOLEAN flag = True;
```

typedef: Examples (2/4)

```
typedef char* STRING;

STRING names[3];

names[0] = "Clint";
names[1] = "Eastwood";
```

typedef: Examples (3/4)

```
// in header file
struct date_tag {
   int day;
   int month;
   int year;
};
typedef struct date_tag DATE;
typedef DATE* DATEPTR;

// in source file
DATE bday = {25, 12, 2019};
```

typedef: Examples (4/4)

```
// in header file
typedef struct {
   int day;
   int month;
   int year;
} DATE;
typedef DATE * DATEPTR;

// in source file
DATE bday = {25, 12, 2019};
```

Trailing Return Type Syntax

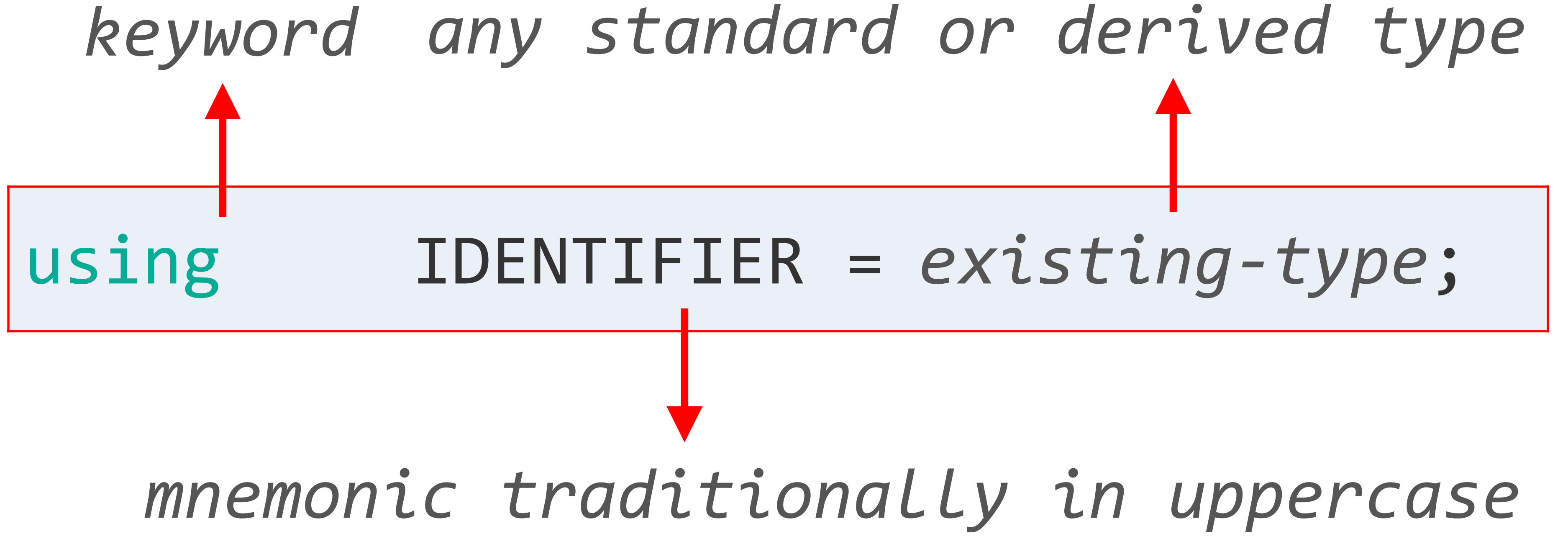
- Trailing return type follows parameter list and is preceded by `->`
- To signal return follows parameter list, use `auto` where return type ordinarily appears
- Given ordinary declaration

```
    char *(*foo(char *, int))[5];
```

can simplify using trailing return type syntax:

```
auto foo(char *, int) -> char* (*) [5];
```

`using`: Alias Declaration Syntax



keyword
any standard or derived type
using IDENTIFIER = *existing-type*;
mnemonic traditionally in uppercase

using: Examples

```
// works in both C and C++ code
typedef int64_t BigInt;
typedef int32_t (*PtrToFunc)(double);

// works since C++11
using BIGINT     = int32_t;
using PTRTOFUNC = int32_t (*) (double);
using PTR_FUNC2 = auto (*)(double) -> int;
```