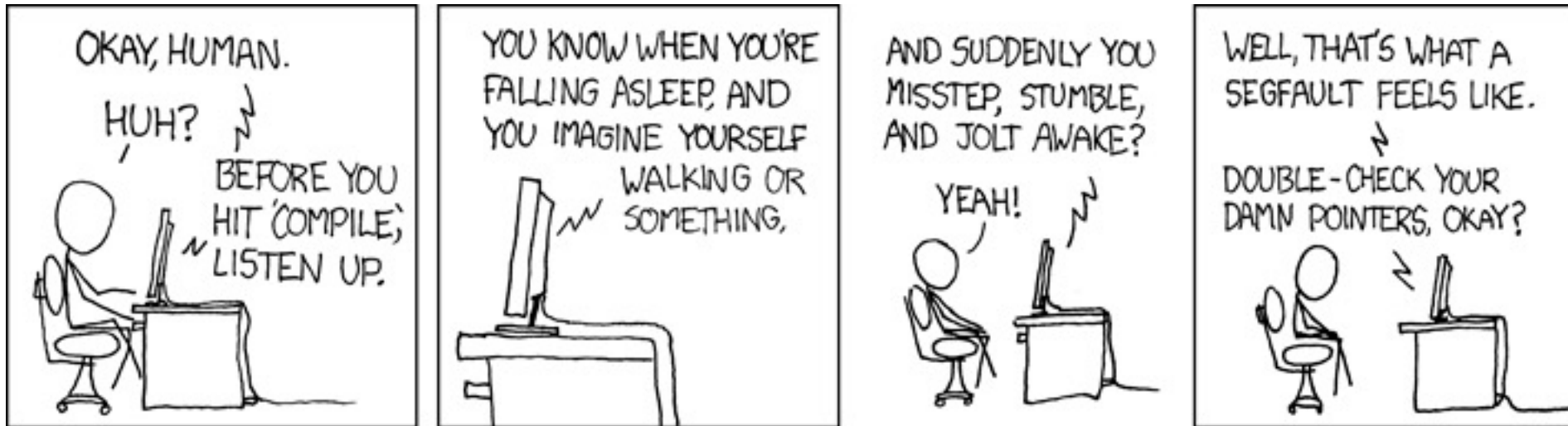


VE280 Programming and Elementary Data Structures

Paul Weng
UM-SJTU Joint Institute

`const` Qualifier; `typedef`



Announcements

- OH starts this week
 - Tue and Mon 14:30-15:30 @ JI 406
 - Mon 20:00-22:00 (Yanjun Chen)
 - Wed 11:00-13:00 (Jiayao Wu)
 - Wed 20:00-22:00 (Chengsong Zhang)
- Lab 1 released (due May 25 23:59)
- P1 released (due May 27 23:59)
- Make-up classes
 - Jun 1 => Jul 2 14:00-15:40
 - Jun 3 => Prof. Weikang Qian

Learning Objectives

- Understand what is the const qualifier and when to use it
 - Know what a const reference is
 - Know the difference between a const pointer and a pointer to a const
- Know how to use typedef

const Qualifier

- Often, a numerical value in a program could have some valid meaning.

```
char name[256];
```

The max size of name string

- Also, that value with the same meaning may appear many times in the program

```
for (i=0; i < 256; i++) ...
```

- If we only use 256, it has two drawbacks
 - The readability is bad.
 - If we need to update max size of a name string from 256 to 512, we need to examine each 256 (some may have other meanings) and update the corresponding ones.
 - It takes time and is error-prone!

const Qualifier

- Instead of just using 256, define a constant, and use the constant:

```
const int MAXSIZE = 256;  
char name[MAXSIZE];
```

- Usually, constant is defined as a global variable.
- Property
 - Cannot be modified later on
 - Must be initialized when it is defined

```
const int a = 10;  
a = 11; // Error
```

```
const int i;  
// Error
```

const Reference

```
const int iVal = 10;  
const int &rVal = iVal;
```

- Furthermore, const reference can be initialized to an rvalue

```
const int &ref = 10; // OK  
const int &ref = iVal+10; // OK
```

- In contrast, nonconst reference cannot be initialized to an rvalue

```
int &ref = 10; // ERROR  
int &ref = iVal+10; // ERROR
```

Practical Use of const Reference

- One popular use of const reference: pass struct/class as the function argument

```
int avg_exam(const struct Grades & gr) {  
    return (gr.midterm+gr.final)/2;  
}
```

- In comparison:

```
int avg_exam(struct Grades gr) { ... }
```

Problem? Pass-by-value can be **expensive**,
particularly for large structures.

```
int avg_exam(struct Grades & gr) { ... }
```

Problem? It allows for the possibility of (mistakenly)
changing the contents of the **caller's** `gr`.

Practical Use of const Reference

- One popular use of const reference: pass struct/class as the function argument

```
int avg_exam(const struct Grades & gr) {  
    return (gr.midterm+gr.final)/2;  
}
```

- Advantages of using const reference as argument
 - We don't have the expense of a copy.
 - We have the safety guarantee that the function cannot change the caller's state.

Practical Use of const Reference

- Compared with non-const reference, another advantage is function call with consts or expressions is OK
 - In contrast, for non-const reference, function call with consts or expressions is not OK

```
foo("Hello world!")
```

```
void foo(string & str) {...}
```

versus

```
void foo(const string &str) {...}
```

const Pointers

- When you have pointers, there are two things you might change:
 1. The value of the pointer.
 2. The value of the object to which the pointer points.
- Either (or both) can be made unchangeable:

```
const T *p;    // "T" (the pointed-to object)
               // pointer to const // cannot be changed by pointer p
T *const p;    // "p" (the pointer) cannot be
               // const pointer  // changed
const T *const p; // neither can be changed.
```

Pointers to const

Example

```
int a = 53;
const int *cptr = &a;
    // OK: A pointer to a const object
    // can be assigned the address of a
    // nonconst object
*cptr = 42;
    // ERROR: We cannot use a pointer to
    // const to change the underlying
    // object.
a = 28 // OK
int b = 39;
cptr = &b; // OK: the value in the pointer
           // can be changed.
```

const Pointers

Example

```
int a = 53;
int *const cptr = &a;
    // OK: initialization
*cptr = 42;
    // OK: We can use a const pointer to
    // change the underlying object.
int b = 39;
cptr = &b;
    // ERROR: We cannot change the value of
    // a const pointer.
```

Define Pointers to const Using typedef

- Recall typedef: gives an alias to the existing types:
`typedef existing_type alias_name;`
- Example: `typedef int * intptr_t;`
Then we can use it: `intptr_t ip;`
- Use `typedef` to define pointer to const:
 - `typedef const T constT_t;`
`typedef constT_t * ptr_constT_t;`
 - Now `ptr_constT_t` is an alias for the type of
`const T *` pointer to const

? How do we use `typedef` to rename the type of `T *const`? const pointer

Select all the correct answers.

- **A.** `typedef const T const_t;`
`typedef const_t *constptrT_t;`
- **B.** `typedef T *ptrT_t;`
`typedef ptrT_t const constptrT_t;`
- **C.** `typedef const * constptr_t;`
`typedef constptr_t T constptrT_t;`
- **D.** `typedef T * const constptrT_t;`



Practical Use of Pointer to const

Example

```
void strcpy(char *dest, const char *src)
    // src is a NULL-terminated string.
    // dest is big enough to hold a copy of src.
    // The function place a copy of src in dest.
    // src is not changed.
{ ... }
```

- Strictly speaking, we don't **need** to include the `const` qualifier here **since the comment promises that we won't modify the source string**
- So, why include it?

Practical Use of Pointer to `const`

Example


- Why include `const`?
- Because once you add it, you CANNOT change `src`, even if you do so by mistake.
- Such a mistake will be caught by the **compiler**.
 - Bugs that are detected at compile time are among the easiest bugs to fix – those are the kinds of bugs we want.
- **General guideline**: Use `const` for things that are passed by reference, but won't be changed.

Pointer to const versus Normal Pointer

- Pointers-to-const-T are **not the same type** as pointers-to-T.
- You can use a pointer-to-T anywhere you expect a pointer-to-const-T, but NOT vice versa.


```
int const_ptr(const int *ptr)
{
    ...
}

int main()
{
    int a = 0;
    int *b = &a;
    const_ptr(b);
}
```



```
int nonconst_ptr(int *ptr)
{
    ...
}

int main()
{
    int a = 0;
    const int *b = &a;
    nonconst_ptr(b);
}
```



Pointer to const versus Normal Pointer

- Why can you use a pointer-to-T anywhere you expect a pointer-to-const-T?
 - Code that expects a pointer-to-const-T will work perfectly well for a pointer-to-T; it's just guaranteed not to try to change it.
- Why **cannot** you use a pointer-to-const-T anywhere you expect a pointer-to-T?
 - Code that expects a pointer-to-T might try to change the T, but this is illegal for a pointer-to-const-T!



Variable `x` is declared as being of type

`const T * const.`

Select all the correct answers.

- **A.** My code cannot change `x` (using `x`).
- **B.** My code cannot change the value pointed by `x` (using `x`).
- **C.** `x` may change while the program is running.
- **D.** The value pointed by `x` may change while the program is running.



Reference

- **const Qualifier**
 - C++ Primer, 4th Edition, Chapter 2.4
- **const Pointers**
 - C++ Primer, 4th Edition, Chapter 4.2.5
- **const References**
 - C++ Primer, 4th Edition, Chapter 2.5