

Object-Oriented Programming and Data Structures

COMP2012: Const-ness

Prof. Brian Mak

Prof. C. K. Tang

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China



- `const`, in its simplest usage, is to express a **user-defined constant** — a value that can't be changed.

```
const float PI = 3.1416;
```

- Some people like to write `const` identifiers in **capital letters**.
- In the old days, constants are defined by the `#define` preprocessor directive:

```
#define PI 3.1416
```

Question: Any shortcomings?

- `const` actually may be used to represent more than just numerical constants, but also **const objects**, **pointers**, and even **member functions**!
- The `const` keyword can be regarded as a safety net for programmers: If an object **should not change**, make it `const`.

Example: Constants of Basic Types

```
#include <iostream>                                     /* File: const-basic-grades.cpp */
#include <cstdlib>
using namespace std;

int main( )
{
    const int NUM_STUDENTS = 117;
    const int MAX_GRADE = 100;
    int grade[NUM_STUDENTS];

    // Seed the random number generator by current time
    srand(time(0));

    // Assign grades randomly
    for (int i = 0; i < NUM_STUDENTS; ++i)
        grade[i] =
            static_cast<double>(rand())/RAND_MAX*MAX_GRADE + 0.5;

    for (int i = 0; i < NUM_STUDENTS; ++i)
        cout << "grade of the " << i+1 << "th student = " << grade[i] << endl;

    return 0;
}
```

Example: Constant Objects

```
class Date                                     /* File: const-basic-date.cpp */
{
    int year, month, day;
public:
    Date(int year, int month, int day);
    int difference(const Date&);
    void add_month() { month += 1; };
};

int main( )
{
    const Date job_start(1998, 4, 1);
    int this_year, this_month, today;
    cin >> this_year >> this_month >> today;
    Date today(this_year, this_month, today);

    // How long have I worked in UST in days?
    cout << "I have worked " << today.difference(job_start) << " days.\n";

    // What about next month?
    job_start.add_month( ); // Error caught by compiler; -> today.add_month( )
    cout << today.difference(job_start) << " days by next month\n";
    return 0;
}
```

const and const Pointers

- When a pointer is used, two objects are involved:
 - the **pointer itself**
 - the **object** being pointed to
- The syntax for pointers to constant objects and constant pointers can be confusing. The rule is that
 - any **const** to the **left** of the * in a declaration refers to the **object** being pointed to.
 - any **const** to the **right** of the * refers to the **pointer itself**.
- It can be helpful to read these declarations from **right to left**.

```
/* File: const-char-ptrs1.cpp */  
char c = 'Y';  
char *const cpc = &c;  
char const* pcc;  
const char* pcc2;  
const char *const cpcc = &c;  
char const *const cpcc2 = &c;
```

Example: const and const Pointers

```
#include <iostream>                                     /* File: const-char-ptrs2.cpp */
using namespace std;

int main( )
{
    char s[ ] = "COMP2012";                               // char *const s = "COMP2012"
    char p[ ] = "MATH1013";                               // char *const p = "MATH1013"

    const char* pcc = s;                                   // Pointer to constant char
    pcc[5] = '5';                                          // Error!
    pcc = p;                                              // OK, but what does that mean?

    char *const cpc = s;                                   // Constant pointer
    cpc[5] = '5';                                          // OK
    cpc = p;                                              // Error!

    const char *const cpcc = s;                           // const pointer to const char
    cpcc[5] = '5';                                          // Error!
    cpcc = p;                                              // Error!
}
```

const and const Pointers ..

Having a pointer-to-const pointing to a non-const object doesn't make that object a constant!

```
/* File: const-int-ptr.cpp */
```

```
int i = 151;
```

```
i += 20; // OK
```

```
int* pi = &i;
```

```
*pi += 20; // OK
```

```
const int* pic = &i;
```

```
*pic += 20; // Error! Can't change i through pic
```

```
pic = pi; // OK
```

```
*pic += 20; // Error! Can't change *pi thru pic
```

```
pi = pic;
```

```
// Warning: assignment to 'int *' from 'const int *' discards const
```

const: References as Function Arguments

- There are 2 good reasons to pass an argument as a **reference**. What are they?
- You can (and should!) express your intention to leave a reference argument of your function **unchanged** by making it **const**.
- There are 2 advantages:

1. If you **accidentally** try to **modify** the argument in your function, the compiler will catch the error.

```
void cbr(Larg_Obj& LO) { LO.height += 10; } // Fine
```

```
void cbcr(const Larg_Obj& LO) { LO.height += 10; } // Error!
```


const: References as Function Arguments ..

2. You can call a function that has a **const reference parameter** with both **const** and non-**const** arguments.

Conversely, a function that has a non-**const** reference parameter can only be called with non-**const** arguments.

```
void cbr(Larg_Obj& LO) { cout << LO.height; }  
void cbr(const Larg_Obj& LO) { cout << LO.height; }
```

```
int main( )  
{  
    Large_Obj dinosaur(50);  
    const Large_Obj rocket(100);  
    // Which of the following give(s) compilation error?  
    cbr(dinosaur);  
    cbr(rocket);  
    cbr(dinosaur);  
    cbr(rocket);  
}
```

Pointer vs. Reference

Reference can be thought as a special kind of pointer, but there are 3 big differences:

- 1 A pointer can point to nothing (NULL), but a reference is always bound to an object.
- 2 A pointer can point to different objects at different times (through assignments). A reference is always bound to the same object.

Assignments to a reference does not change the object it refers to but only the value of the referenced object.

- 3 The name of a pointer refers to the pointer object. The * or -> operators have to be used to access the object.

The name of a reference always refers to the object. There are no special operators.

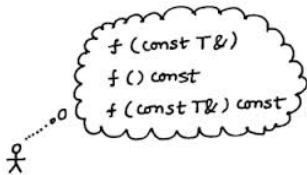
const: Member Functions

To indicate that a **class member function** does **not modify** the class object — its data member(s), one can (and should!) place the **const** keyword **after** the argument list.

```
class Date
{
    int year, month, day;

    public:
        int get_day() const { return day; }    // inline + const
        int get_month() const { return month; } // inline + const
        void add_year(int y);                 // non-const function
};
```

Summary: Good Practice



- Objects you don't intend to change \Rightarrow **const objects**.

```
const double PI = 3.1415927;  
const Date HandOver(1, 7, 1997);
```

- Function arguments you don't intend to change
 \Rightarrow **const arguments**.

```
void print_height(const Large_Obj& LO){ cout << LO.height(); }
```

- Class member functions don't change the data members
 \Rightarrow **const member functions**.

```
int Date::get_day() const { return day; }
```