Comp151

const-ness

const

- <u>const</u>: used in C++ to express a user-defined constant — a value that can't be changed.

```
const float PI = 3.1416;
int = 1;  const int j = 2*i;
```

- Constant variables are usually written in capital letters.
- In the old days, constants were defined by the #define preprocessor directive:

```
#define PI 3.1416
```

- The const keyword can be regarded as a safety net for programmers. If an object *should* not change, make it a const object; the compiler will issue an error message if you try to change a const object.

Example: Constants of Basic Types

```
#include <iostream.h>

const int i = 3;
const float PI = 3.1416;

void main()
{
  for (int j = 1; j <= i; j++)
        cout << j << "*PI = " << j*PI << endl;
}
```

A const MUST be initialized.

```
const int i;
```

is an error.

Example: Constant Objects

```
class Date
{                                            //not really a legal class definition
    int year, month, day;
    Date(int, int, int);                                     // day, month, year
    int difference(const Date & NewDate);          // NewDate is a const ref param
    void add_month() { month += 1; };
};

int main()
{
    const Date job_start(1,4,1998);
    Date x(6,3,2000);

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

    // What about next month?
    job_start.add_month();                          // Error, but caught by compiler
    cout << "In a month I'll have worked " << x.difference(job_start) << " days.\n";
}
```

const and Pointers

Suppose that

const int i =5; int* pi;

and we were allowed to write

pi = & i;

It would be impossible for the compiler to stop

***pi = 10;**

from changing **i**. This would violate the principle behind **const**.

C++ therefore does not allow a regular pointer to point to a const. Only a special *pointer to a const* can point to a const. If a regular pointer points to a const the compiler will complain.

const int * pi;

Pointer to a const

const int * pi; is a *pointer to a const*.
It is not a pointer which is a const!

- **pi** can point to either a const or a non const
- **pi** can be changed.
- ***pi** can not be used in an assignment.
- Only a special *pointer to a const* can point to a const. If a regular pointer points to a const the compiler will complain.

```
j= 10;  const int i = 5;
const int * pi;
pi = &i;  pi = &j;          \\OK pi can change
pi = &i; *pi = 10;          \\error *pi can not be assigned to
pi = &j; *pi = 10;          \\error *pi can not be assigned to
int *qi; qi = &i;           \\error qi is not a pointer to const
```

const and Pointers

We can also have a *pointer that is a constant*. This implies nothing about the item being pointed to.

```
int i = 5;
int * const ri = &i;  \\ const, so must be assigned

cout << *ri;          \\OK
*ri = 10;             \\OK


int j;
ri = &j;                \\ error changing ri
```

const and Pointers

Finally, we can have *pointer to a constant that is a constant itself*. that is, the pointer can not be changed and the thing it points to can not be changed.

```
const int i = 5;
const int * const ri = &i;
cout << "*ri = " << *ri << endl;     \\OK
*ri = 10;                            \\error
int j; ri = &j;                      \\error
```

Note that such a pointer can point to a non const.
It just can not change it.

```
int k = 5;
const int * const ri = &k;           \\OK
*ri =10;                             \\error
```

const and Pointers

We have just seen three different types of pointers:

1. **const int * pi;**
   A pointer to a constant
2. **int * const ri = &i;**
   A pointer that is a constant
3. **const int * const ri = &i;**
   A pointer to a constant that is a constant itself

The two distinct concepts to keep in mind are

- An object that is a constant can not be changed.
- If **pi** is defined as a *pointer to a const* this means that ***pi** can not be assigned to.

const and Pointers

When using a pointer, two objects are involved: the pointer itself, and the object pointed to.

- The syntax for pointers to constants and constant pointers can be confusing.
  The rule is that any **const** to the left of the * in a declaration refers to the object 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.

```
char c = 'Y';
char *const cpc = &c;
const char* pcc;
const char *const cpcc = &c;
```

const: References as Function Arguments

While there are 2 good reasons (what are they?) to pass an argument as a reference, you can (and should!) express your intention to leave a reference argument of your function unchanged by making it **const**. This has 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 cbcr(const Larg_Obj& LO) { cout << LO.height; }

int main() {
  Large_Obj dinosaur(50);
  const Large_Obj rocket(100);

  cbr(dinosaur);
  cbcr(dinosaur);
  cbr(rocket);                                    // Error
  cbcr(rocket);
}
```

const: Member Functions

To indicate that a class member function does not modify the class object, 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; }
    int get_month() const { return month; }
    void add_year(int y);                  // Non-const function
};
```

Summary

It is good practice to make:

- objects that you don't intend to change **const**.

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

- function arguments that you don't intend to change **const**.

  void print_height(**const** Large_Obj& LO){ cout ≪ LO.height(): }

- class member functions that do not change the object **const**.

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