Comp151

Const-ness

Watch out!

- The keyword `const` has many different meanings in C++, depending on where it's used.

const

- `const` in variable declarations: used 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 bad old days, constants were defined by the ugly `#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>
using namespace std;

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: the following is an error!

```
const int i;    // will give a compile-time error
```

Example: Constant Objects

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

int main()
{
    const Date job_start(1, 10, 1992);
    Date x(27, 2, 2006);

    // How long have I worked at UST in days?
    cout << 'I have worked for " << 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;            // actually, this is illegal
  ```

- Then 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;
  pi = &i;            // now this is ok
  ```

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` cannot be changed, i.e., it cannot be used in an assignment.
  - Only a special pointer to a const can point to a const. If you try to set a regular pointer to point to a const the compiler will complain.

```
int 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 cannot be assigned to (even though j can)
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;                     // compile-time error: cannot change ri
```

const and Pointers

- Finally, we can have both: a pointer to a constant that is also a constant itself. That is, the pointer cannot be changed and the thing it points to also cannot be changed.

```
const int i = 5;
const int* const ri = &i;
cout << "*ri = " << *ri << endl;        // ok
*ri = 10;                               // compile-time error
int j; ri = &j;                         // compile-time 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;                                // compile-time 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 cannot 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 very 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 and References

- The syntax for references that refer to constants is just like the syntax for pointers that refer to constants, and the rules are the same:

```
char c = 'Y';
const char& rcc = c;
c = 'X';
rcc = 'Z';                  // compile-time error
```

- But unlike pointers, there is no need for references that are themselves constants. (Why?)

const and References: 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(LargeObj& a)
{
   a.height += 10;                    // no compile-time error
}
void cbcr(const LargeObj& a)
{
   a.height += 10;                    // compile-time error!
}
```

const and References: References as Function Arguments (cont)

2. You can call a function that has a const reference parameter with either const and non-const arguments. But a function that has a non-const reference parameter can only be called with non-const arguments.

```
void cbr(LargeObj& a) { cout << a.height; }
void cbcr(const LargeObj& a) { cout << a.height; }

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

   cbr(dinosaur);
   cbcr(dinosaur);
   cbr(rocket);                          // compile-time 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 {
public:
   int get_day() const { return day; }
   int get_month() const { return month; }
   void add_year(int y);                    // non-const function
private:
   int year, month, day;
};
```

- For an acceptable software engineering standard, you should always follow this practice to maintain **const correctness** (even though you might find it much easier to compile working programs without doing so!) This way the compiler can help catch bugs before they do any damage.

Summary

- Acceptable software engineering practice demands that you make the following `const`:
  - objects that you don't intend to change

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

  - function arguments that you don't intend to change

```
void print_height(const LargeObj& a) { cout << a.height(); }
```

  - class member functions that do not change the object

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