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

```cpp
#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: the following is an error!

    const int i;   // will give a compile-time error
```
Example: Constant Objects

class Date \hspace{1cm} // not really a complete class definition
{
    int year, month, day;
    Date(int, int, int); \hspace{1cm} // day, month, year
    int difference(const Date & NewDate); \hspace{1cm} // 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 at UST in days?
    cout << "Today I have worked " << x.difference(job_start) << " days.\n";

    // What about next month?
    job_start.add_month(); \hspace{1cm} // 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

• \texttt{const int * pi;} is a \textit{pointer to a const}. It is not a pointer which is a const!
  - \texttt{pi} can point to either a const or a non const.
  - \texttt{pi} can be changed.
  - \texttt{*pi} cannot be changed, i.e., it cannot be used in an assignment.
  - Only a special \textit{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.

\begin{verbatim}
int j = 10; const int i = 5;
const int * pi;
pi = &i; pi = &j; \hspace{1em} // ok: \texttt{pi} can change
pi = &i; *pi = 10; \hspace{1em} // error: \texttt{*pi} can not be assigned to
pi = &j; *pi = 10; \hspace{1em} // error: \texttt{*pi} cannot be assigned to (even though \texttt{j} can)
int *qi; qi = &i; \hspace{1em} // error: \texttt{qi} is not a pointer to const
\end{verbatim}
const and Pointers

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

```c
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.

```cpp
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.

```cpp
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` cannot 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 * 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: 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:
   ```c++
   void cbr(Large_Obj& LO)
   {
       LO.height += 10; // ok
   }
   void cbcr(const Large_Obj& LO)
   {
       LO.height += 10; // compile-time error!
   }
   ```
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.

```c
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); // 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.

```cpp
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

• Acceptable software engineering practice demands that you 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; }