G52CPP
C++ Programming
Lecture 11

Dr Jason Atkin

This lecture

- const

- Constants
- const pointers
- const references
- const member functions

const

Defensive programming

const : constant/unchanging

- **`const`**ant *variables* cannot be changed
- E.g. `const int maxvalue = 4;`
- Or `int const maxvalue = 4;`

- Not really '*variable*'s anymore? Cannot be 'varied'

- **`#define`** could have same effect – see later
  - But, using **text replacement** in the **preprocessor**
- **`const`** is nicer for declaring constants
  - Multiple contradictory definitions will be caught
  - Unlike for **`#define`**

Pointers to constant data

- The thing pointed at through a **pointer to const** cannot be changed **using the pointer**
- E.g. `const char* p = “Hello”;`
- Or `char const* p = “Hello”;`

- **Note: `const` is to the left of the `*`**

- The following code will NOT compile:

```
const char* pc = "Hello";
*pc = 'B'; // BAD
```

- String literals should be `const char*` not `char*` and good compilers will ensure this (warnings)

Constant pointers

- You can also prevent the pointer itself from being changed, by using **`const`**. E.g.:

```
char* const p = “Hello”;
```

Note: the **`const`** is to the **right** of the **`*`**

- You cannot change this **pointer** to make it point at something else
- The following code will **not compile**:

```
char* const cp = "Hello";
cp = "Bye"; // BAD
```

– i.e. catch errors at compilation!

For pointers, it matters where the const is

For constant pointers it matters which side of the `*` the `const` is:

- The **pointer** is constant – constant `short*` :

```
short * const pcs = &s;
```

- The short **pointed at** cannot be changed through the pointer – pointer to constant `short` :

```
short const * cps = &s;
const short * cps = &s;
```

- Can change neither pointer nor thing pointed at :

```
short const * const cpcs = &s;
const short * const cpcs = &s;
```

How to remember this…

- Read backwards with **`*`** meaning ‘pointer to’

`float * const pcf = &f;`

  – “Constant pointer to a float”

The **pointer** is constant – constant **`float*`**

`float const * cpf = &f;`

  – “Pointer to constant float”

`const float * cpf = &f;` (same as **`float const *`**)

  – “Pointer to float which is constant”

The float **pointed at** cannot be changed through the pointer

`const float * const cpcf = &f;`

  – “Constant pointer to float which is constant”

Neither the pointer nor the **thing it points at** can be changed

String literals again

- String literals should not be changed
- i.e. use **`const`** pointers

- Should use:

```
const char* str = “Hello”;
```

- Not:

```
char* str = “Hello”;
```

- Compiler should give warnings otherwise

`const` references/pointers to objects

`const` references

- `const` references make the thing referred to const
  - `const` for pointers can mean either ***unchangable pointer*** or ***the thing pointed at cannot be changed***
  - You cannot make a reference refer to something else anyway, so `const` always means the thing referred to

- `const` references are useful for parameters
  - Passing by value (not reference) means the original variable cannot be accidentally modified
    - May be safer
  - Passing a reference means that no copy is made
    - May be quicker – copying objects can be slow
  - Using a `const` reference means no copy needs to be made, but the original can still not be changed, **like a copy but faster**

`const` references and pointers

- Q: If you have a `const` reference (or pointer) to an object, then which ***methods*** can you call using the reference (or pointer)?

```
MyClass ob2;
const MyClass& rob2a = ob2;

rob2a.GetVal(); // ?
rob2a.SetVal(); // ?
```

`const` references and pointers

- Q: If you have a **`const`** reference (or pointer) to an object, then which ***methods*** can you call using the reference (or pointer)?
- A: Only methods which **guarantee** not to change the object (i.e. accessors)
- **These methods are labelled `const`**
  - **They CANNOT alter member data**
  - **The this pointer is const**
- Functions are either mutators or accessors
  - Accessors only access data – should be **`const`**
  - Mutators change data – **cannot** be **`const`**

Which of these lines will not compile?

```
class ConstClass
{
public:
   // Constructor
   ConstClass()
   {}

   // Accessor
   int GetVal() const
   { return _ival; }

   // Mutator
   void SetVal(int ival)
   { _ival = ival; }

private:
   int _ival;
};
```

```
int main()
{
   ConstClass ob2;
   ConstClass& rob2 = ob2;
   const ConstClass& rob2a = ob2;
   ConstClass const& rob2b = ob2;

   rob2.GetVal();
   rob2a.GetVal();
   rob2b.GetVal();

   rob2.SetVal(3);
   rob2a.SetVal(1);
   rob2b.SetVal(2);
}
```

Example: `const` functions

```
class ConstClass
{
public:
   // Constructor
   ConstClass()
   {}

   // Accessor
   int GetVal() const
   { return _ival; }

   // Mutator
   void SetVal(int ival)
   { _ival = ival; }

private:
   int _ival;
};
```

```
int main()
{
   ConstClass ob2;
   ConstClass& rob2 = ob2;
   const ConstClass& rob2a = ob2;
   ConstClass const& rob2b = ob2;

   rob2.GetVal();
   rob2a.GetVal();
   rob2b.GetVal();

   rob2.SetVal(3);

   // The following 2 lines
   // do not compile
   rob2a.SetVal(1);
   rob2b.SetVal(2);
}
```

mutable

mutable

- The compiler will **not allow** you to alter member data from a member function declared as **`const`**
  - If you try, then you will get a compilation error
- If you need to alter a **specific** variable within a **`const`** member function, you can declare **that variable** **`mutable`**
- e.g. for a class which caches the last value retrieved:

```
class CachingClass
{
  int _iVal;
  mutable int _lastgot;
public:
  int GetVal() const
        {_lastgot = _iVal; return _iVal; }
  void SetVal( int iVal ) const
        { _iVal = iVal; }
}
```

This can be altered even by **`const`** member functions

mutable

- The compiler will **not allow** you to alter **any** member data from a member function declared as **`const`**
  - If you try, then you will get a compilation error
- If you need to alter a specific variable within a **`const`** member function, you can declare that variable **`mutable`**
- e.g. for a class which caches the last value retrieved:

```
class CachingClass
{
  int _iVal;
  mutable int _lastgot;
public:
  int GetVal() const
        {_lastgot = _iVal; return _iVal; }
  void SetVal( int iVal ) const
        { _iVal = iVal; }
}
```

OK, since **`_lastgot`** is mutable

Compilation error **`const`** fn sets **`_ival`**

`const` members

`const` member data

```
class DemoClass
{
public:
  DemoClass()
  : ci(4)
  , cj(12)
  {}
private:
  int const ci;
  const int cj;
};
```

- **`const` member** data **MUST** be initialised in the initialisation list for the constructor
  - i.e. an initial value when member data is constructed
- Cannot just be set in constructor body, since construction has occurred by then
- **Compiler error if you miss any**

Note: Relative order of **`const`** and type only matters for pointers **`const *`** vs **`* const`**

Visual Studio

Getting started

Running a program

Debugging / break points

Next Lecture

- Namespaces and scoping
- Some standard class library classes
  - String
  - Input and output