# Types and Default Class Members



RMIT UNIVERSITY

# Parameter Passing

- Several different mechanisms are used by programming languages to pass arguments to functions
- The most common are:
  - Pass by value - a copy of the actual argument is passed
  - Pass by reference - the address of the actual argument is passed
- Java uses pass by value for primitive types, and pass by reference for objects
  - C uses pass by value for everything
    - But pass by reference can be implemented in C by using pointer types as arguments
- Pass by value is inefficient for large objects

# Parameter Passing (cont.)

- Which `Movie` member function(s) use inefficient copies for arguments
  - And how can we improve the efficiency of argument passing in `Movie`?

```
// File "movie1.h"
#include "clip1.h"
#include <vector>
typedef std::vector<Clip*> ClipSeq;
class Movie : public Video {
  ClipSeq seq;
public:
  bool play();
  // play each clip in turn
  Clip* find(std::string info);
  // the first Clip with info
  // that matches info
  bool remove(Clip* cp);
  void add(Clip* cp);
  Movie() : seq() {}
};
```

NOTE: this default constructor is optional, as the compiler will generate it if there are no other constructors, because a member, `seq`, has a default constructor

# Parameter Passing (cont.)

- `Movie::find` passes a `std::string` by value
  - it would be more efficient to pass it using a pointer

```
// File "movie2.h"
#include "clip1.h"
#include <vector>
typedef std::vector<Clip*> ClipSeq;
class Movie : public Video {
  ClipSeq seq;
  ...
  Clip* find(std::string* info);
  ...
```

```
// File "movietest2.cpp"
#include <iostream>
#include "movie1.h"
int main() {
  ...
  Movie m;
  ...
  std::string s("Sink");
  Clip* cp = m.find(&s);
  ...
}
```

```
// File "movie2.cpp"
...
Clip* Movie::find(std::string* info) {
// return the first Clip with info that includes info, or NULL
  ...
  for (ClipSeq::iterator it = seq.begin(); it != seq.end(); ++it)
    if ((*it)->get_info().find(*info) != string::npos)
      ...
```

Changes in 3 files underlined

# Parameter Passing (cont.)

- Passing arguments using pointers is tedious and error-prone
  - The formal parameter must be declared as a pointer in the function declaration, then dereferenced in the function body
  - The actual parameter to the function call must be an address
- C or C++ programs with many `*`'s and `&`'s are hard to read and maintain
- C++ provides reference parameters as abbreviations for explicit pointers

# References

- C++ provides explicit reference types, declared using `&`

```
// File "movie3.h"
#include "clip1.h"
#include <vector>
typedef std::vector<Clip*> ClipSeq;
class Movie : public Video {
  ClipSeq seq;
  ...
  Clip* find(std::string& info);
  ...
```

```
// File "movietest3.cpp"
#include <iostream>
#include "movie1.h"
int main() {
  ...
  Movie m;
  ...
  Clip* cp = m.find("Sink");
  ...
}
```

To convert value type parameters to reference types ONLY the function specification needs to change

```
// File "movie2.cpp"
...
Clip* Movie::find(std::string& info) {
  // return the first Clip with info that includes info, or NULL
  ...
  for (ClipSeq::iterator it = seq.begin(); it != seq.end(); ++it)
    if ((*it)->get_info().find(info) != string::npos)
      ...
```

# References (cont.)

- C++ references are very much like Java's pass by references for objects
  - Except that in C++ the "`&`" is needed to make pass by reference explicit
- The symbols "`&`" and "`*`" have uses in both declarations and expressions

# References (cont.)

| Symbol | & | * |
|---|---|---|
| Declaration | "reference to" | "pointer to" |
| Expression | "take address of" | "dereference" or "follow the pointer" |

- Pass by reference works with virtual functions

```
bool transfer(Account& from, Account& to, Money amount) {
  // transfer money between any Account types
  // using virtual functions withdraw and deposit
  if (!from.withdraw(amount)) return false;
  return to.deposit(amount);
}
```

Can pass in FlexiAccount, SuperFlexiAccount, CheckingAccount, etc.

# References (cont.)

- Our `Video` classes currently return
  - Status values, such as `bool` for `Video::play()`
  - Object copies, such as `std::string` for `Clip::get_info()`
  - A pointer to an object, such as `Clip*` for `Movie::find()`
- In C++, a function can return an object by using a reference result
  - So we could return `std::string&` for `Clip::get_info()` instead of `std::string`
    - Avoiding an inefficient copy of an object

# References (cont.)

- The Clip class now uses references for
  - Returning Clip::info from get_info
  - Passing constructor arguments

```
// File "clip2.h"
#include "video1.h"
#include <string>
class Clip : public Video {
  std::string filename;
  std::string info;
public:
  bool play();
  std::string& get_info()
    { return info; }
  Clip(std::string& init_filename,
       std::string& init_info)
: filename(init_filename), info(init_info) {}
};
```

```
// File "movietest2.cpp"
...
#include "clip2.h"
int main() {
  std::string the_file("launch.mpg");
  std::string the_info("Launch boat");
  Clip c1(the_file, the_info);
  ...
  the_file = "rubbish";
  std::cout << c1.get_info();
  ...
```

Output the object the_info: "launch boat"

# References (cont.)

- Passing and returning objects by reference can be unsafe
  - It gives you a “handle” to an object
- Using const can prevent “accidental” changes to to objects

```
// File "movietest2.cpp"
...
#include "clip2.h"
int main() {
  std::string the_file("launch.mpg");
  std::string the_info("Launch boat");
  Clip c1(the_file, the_info);
  ...
  c1.get_info() = "something";
  ...
```

Changes c1.info to: "something"

# References (cont.)

- Predefined meanings for operators in C and C++ often return objects
  - For fundamental types such as `int`, "`b = a`" returns "`b`"; "`++a`" returns "`a`"
  - This allows for concatenation, such as: "`c = b = a`" or "`b = ++a`"
- Library-defined meanings for operators also return objects

`cout << "The value of x is: "` returns `cout`

# References (cont.)

- The result of the `operator<<` is a reference to an `ostream` object: `cout`, NOT a copy of an object... Why not?
- Suppose we wanted to allow concatenation of `Movie::add()`, e.g

```
movie1.add(clip1).add(clip2)
```

  – What would `Movie::add()` need to return?

# References (cont.)

- A sequence of I/O operations apply to the same object, not a copy:

```
cout << a << b << c << …
```

  – The objects `a`, `b`, and `c` are all output to the one object `cout`
  – If `operator<<` returned an object by value, it would be putting the objects onto different output files - copies of `cout`
- For concatenation, `Movie::add()` needs to return itself: `*this`

# References (cont.)

```
...
typedef std::vector<Clip*> ClipSeq;
class Movie : public Video {
  ClipSeq seq;
public:
  bool play();
  Clip* find(std::string& info);
  Movie& remove(Clip& c);
  Movie& add(Clip& c);
  Movie() : seq() { }
};
```

```
...
Movie& Movie::add(Clip& c) {
  seq.push_back(&cp);
  return *this;
}
```

insert the address of c onto the container seq

Now add and remove can be concatenated e.g., movie1.add(clip1).remove(clip2)

# References (cont.)

- Why not just use references everywhere instead of pointers?

```
...
typedef std::vector<Clip&> ClipSeq;
class Movie : public Video {
  ClipSeq seq;
public:
  bool play();
  Clip& find(std::string& info);
  Movie& remove(Clip& c);
  Movie& add(Clip& c);
  Movie() : seq() { }
};
```

Use a container of references;
find returns a reference to a Clip

```
// File "movie3.cpp"
...
Clip& Movie::find(std::string& info) (
// return the first Clip or a Null
  Clip result = ????; // The Null Clip
  for (ClipSeq::iterator it =
        seq.begin(); it != seq.end();
        ++it)
    if ((*it)->get_info().find(info)
          != string::npos)
        result = *it;
  return result;
}
```

Modify the implementation of Movie::find

# References (cont.)

- The declaration of `std::vector<Clip&>` is a syntax error
- Standard containers cannot have reference types as template arguments
  - Because containers call the “default constructor” for the type they contain, when the container needs to grow
  - But reference objects must be initialized when they are allocated

```
...
Clip c1;
Clip& c_ref1 = c1; // OK
Clip& c_ref1;  // Syntax error
... new Clip&; // Syntax error
```

# References (cont.)

- Also, there is no built-in null reference object
  - So `Movie::find()`, or preferably the `Clip` class, needs to define a "`Null_Clip`" object
- **References are not a simple alternative to pointers**
- **Mixing references and pointers can be confusing**

# Constants

- Many programming languages support declaring constants
  - Java supports final declarations
  - C programmers often use #defines instead

```
#define MAX_CLIPS 42
// substitute 42 for MAX_CLIPS
```

- **`#defines` are not type checked and have no syntactic scope**
- **Use `consts` instead of `#defines` whenever possible**

# Constants (cont.)

- C++ generalizes the notion of something being constant to mean *read-only*
  - Read-only data can include member data and functions as well as global constants, function arguments, and return results
  - Declaring data as read-only can significantly improve program readability, reliability, and efficiency

# Constants (cont.)

- `const`s initialized to constant expressions, such as `MAX_CLIPS`, need not occupy storage
  - Thus `const`s are just as efficient at `#define`s
- A `const` definition requires an initial value, and a `const` cannot be re-assigned or modified

Syntax error: SIZE1

```
// File "module1.cpp"
const int SIZE1;
const int SIZE2 = 2;
...
```

extern declaration references a const in another compilation unit, initial value not legal here

```
// File "module2.cpp"
#include <iostream>
extern const int SIZE2;

int main() {
  std::cin >> SIZE2;
  SIZE2 *= 2;
}
```

Two syntax errors here

# Constants (cont.)

- Initializing one object using an object defined in another compilation unit (another **.cpp** file) leads to undefined behavior
- As C and C++ do not define the order of initialization across compilation units
  - Within a unit it is the order of declaration
- **Beware of cross-compilation unit initialization dependencies!**

# Constants (cont.)

SIZE1 and SIZE2 could be initialized in any order, but SIZE3 is initialized using SIZE2

```
// File "module1.cpp"
const int SIZE1 = 2;
...
```

```
// File "module2.cpp"
#include <iostream>
extern const int SIZE1;
const int SIZE2 = SIZE1;
const int SIZE3 = SIZE2+1;
int main() {
  std::cout << SIZE2;
  ...
}
```

Output might be 2 or 0

# Constants (cont.)

- C++'s type-conversion rules enforce *conservation of constness*
- An object declared as constant *can be copied*, but *cannot be modified*, either
  - Directly—by assignment
  - Potentially indirectly—by converting a const to a non-const to make types match in an assignment or initialization

# Constants (cont.)

```
// File "doit1.h"
float doit_val(float);
float doit_ptr(float*);
float doit_ref(float&);
```

```
#include "doit1.h"
const float PI = 3.14159265;
int main() {
  doit_val(PI);
  doit_ptr(&PI);
  doit_ref(PI);
  ...
```

OK

Syntax error:
"cannot convert
`const float*` to `float*`"

Syntax error:
"cannot convert
`const float` to `float&`"

# Constants (cont.)

- Parameters, like local and global objects, can be const

```
#include "doit2.h"
const float PI = 3.14;
float f;
int main() {
  doit_ptrc(&PI);
  doit_refc(PI);
  doit_ptrc(&f);
  doit_refc(f);
  ...
```

```
// File "doit2.h"
float doit_ptrc(const float*);
float doit_refc(const float&);
```

All legal `const` or non-`const` can be passed to a `const`

- "non-const" types can convert to `const` types, but not the other way
- Since almost all functions in OO programs are member functions, a common use of `const` is in member function declarations

# Constants (cont.)

- Using `const`s can make our `Video` classes more reliable and efficient

```
// File "clip-rc1.h"
#include "video1.h"
#include <string>
class Clip : public Video {
  std::string filename;
  std::string info;
public:
  bool play();
  std::string& get_info() { return info; }
  Clip(const std::string& init_filename,
       const std::string& init_info)
     : filename(init_filename), info(init_info) {}
};
```

- Passing by `const` reference is very common in C++
  - The efficiency of passing by reference with the safety of passing by value
- We could declare data members as `const`s too

# Constants (cont.)

```
// File "clip-rc2.h"
#include "video1.h"
#include <string>
class Clip : public Video {
  const std::string filename;
  const std::string info;
public:
  bool play();
  std::string& get_info() { return info; }
  Clip(const std::string& init_filename,
       const std::string& init_info)
    : filename(init_filename), info(init_info) {}
};
```

- The program above contains a syntax error
  - What is it and how can it be fixed?
  - HINT: `get_info()` contains a type conversion error

# Constants (cont.)

- The return type of `get_info()` is `std::string&`
  - But there is no conversion of info, a `const std::string` to `std::string&`
  - That would violate the *conservation of constness*
- To fix the error we must make the return type of `get_info()` a `const`

# Constants (cont.)

```
// File "clip-rc3.h"
#include "video1.h"
#include <string>
class Clip : public Video {
  const std::string filename;
  const std::string info;
public:
  bool play();
  const std::string& get_info() { return info; }
  Clip(const std::string& init_filename,
       const std::string& init_info)
    : filename(init_filename), info(init_info) {}
};
```

```
#include "clip-rc3.h"
...
const Clip clip1("hello.mpg", "Hi!");
Clip clip2("message.mpg", "Help!");

int main {
  ...
  std::cin >> clip2.get_info();
  std::cout << clip1.get_info();
  ...
```

Syntax error here, cannot modify const reference result

Syntax error here too: `clip1.get_info()` Why?

# Constants (cont.)

- We cannot call `get_info()` on the const object clip1
- We must somehow declare that `get_info()` does not modify the object
  - This is done using `const` member functions
- To fix the error we must make the return type of `get_info()` a `const`

# Constants (cont.)

```
#include "clip-rc3.h"
...
const Clip clip1("hello.mpg", "Hi!");
Clip clip2("message.mpg", "Help!");

int main {
   ...
   std::cout << clip1.get_info();
   ...
```

Legal now

```
// File "clip-rc3.h"
#include "video1.h"
#include <string>
class Clip : public Video {
   const std::string filename;
   const std::string info;
public:
   bool play();
   const std::string& get_info() const { return info; }
   Clip(const std::string& init_filename,
        const std::string& init_info)
     : filename(init_filename), info(init_info) {}
};
```

const member function

# Constants (cont.)

- A `const` member functions cannot modify member data
  - Modify member data of the current object directly
  - Call a non-`const` member (that might modify member data)
  - return `*this` as a non-`const` object return type or return a data member as a non-const reference return type

# Constants (cont.)

```
// File "video-c1.h"
class Video {
public:
  virtual bool play()const = 0;
...
```

```
// File "clip-rc4.h"
#include "video-c1.h"
...
class Movie : public Video {
  ClipSeq seq;
public:
  bool play() const;
  ...
```

```
// File "movie1.cpp"
#include "movie1.h"
bool Movie::play() const {
  bool all_play = true;
  for (ClipSeq::const_iterator it
    = seq.begin();
    it != seq.end(); ++it)
  if ((*it)->play())
    all_play = false;
  return all_play;
}
...
```

`const` iterator needed

All 3 play function declarations and definitions must be `const` members

# Constants (cont.)

- Using `const`s on member functions, their arguments, and return types, enforces quite strict discipline on the programmer
  - Making something const in an existing application can have major “ripple effects”
  - To overcome this, there is even a mutable type modifier that will override `const` checking
  - More recent programming languages, such as .NET, have rejected the strict syntactic use of type-checked `const`s

# Strings and Streams

# The string class

- The `std::string` class is a `typedef` for the following *template class*.  (More on templates in Lecture 7)

```
typedef std::basic_string<char> string;
```

- The string class provides a rich set of functionality which we have not fully explored.  We will do so in this lecture.
- The string class is designed to replace C-style strings, and should be preferred over them as much as possible.

# The string class

- We can edit characters within a string using subscripting – like a C-style string:

```
std::string tmp = “Hello”;
tmp[0] = ‘J’;  // single quotes!
cout << tmp << endl; // outputs “Jello”
```

# The string class

- We can initialise a string in a number of different ways:

```
string s = "Hello";
string empty;  // empty string
string s2 = s;  // copy initialisation
string s3(3, 'w'); // 3 of 'w', or "www"
string s4(s, 3, 2); // s[3], s[4], or "lo"
```

# The string class

- To get a C-style string from a string object, use the `c_str()` member function. This string is null terminated.

```
const char* tmp = s.c_str();
```

- For a non-null terminated string, use the `data()` member function.

# The string class

- String comparison can be done using the == operator:

```
if (s1 == s2) { }
```

- Alternatively, you can use a large range of compare functions, e.g.

```
if (s1.compare(0, 3, s2)) {
   // true if s1[0] -> s1[2]
   // compare with s2[0] -> s2[2]
}
```

# The string class

- Text can be inserted onto a string using the `+=` operator, the `append()` member function or the `insert()` member function.

```
string s1 = “Hello”;
s1 += “ World”; // s1 = “Hello World”
s1.append(“!”); // s1 = “Hello World!”
s1.insert(5, “,”); // s1 = “Hello, World!”
```

# The string class

- The + operator performs string concatenation.

```
string s = s1 + s2;
```

- A substring can be found using the `substr()` member function.

```
string s1 = “Hello”;
string s = s1.substr(3); // s = “lo”
string s2 = s1.substr(2, 2); // s = “ll”
```

# The string class

- Finding a string or character within a string is performed with the `find()` functions. There are a large range of find functions, including `find_last_of()`, `find_first_of()`, `find_last_not_of()`, and `find_first_not_of()`.

- If a string is not found, `string::npos` is returned. `string::npos` is the length of the longest possible string.

# The string class

```
std::string s = “Hello, World!”;

int pos = s.find(“Hello”);
int pos2 = s.find_first_of(“He”);
int pos3 = s.find_last_not_of(“lo”);
int pos4 = s.find_last_not_of(‘l’);
int pos5 = s.find_first_not_of(“el”);
int pos6 = s.find_first_not_of(‘H’);
```

# The string class

- We can `replace()` and `erase()` sections of a string.

```
std::string s = “This is a bad test”;
s.replace(s.find(“bad”), 4, “good”);
s.erase(s.find(“good”), 5, “”);
s.erase();  // empty string
```

# The string class

- For further reading, see Stroustrup, Chapter 20.

# Streams



RMIT UNIVERSITY

# Streams

- When discussing input and output, there is often a relationship where data flows from one point to another. The path it flows is called a **stream**.
- We are usually only concerned with the stream's **endpoints**.
- Endpoints could be a file, the screen, or a network host.

# Streams

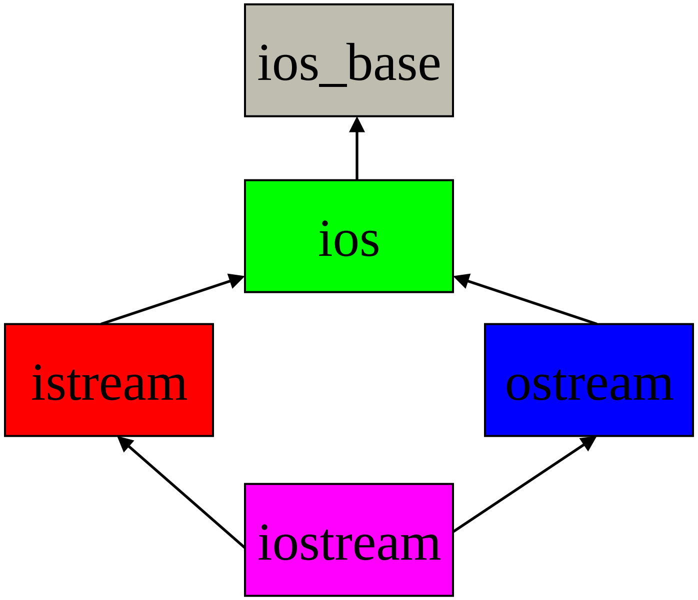
- We have already seen a few streams – `cin` and `cout`.
- `cin` is a stream where the endpoints are the application and the keyboard.
- `cout` is a stream where the endpoints are the application and the screen.
- These are defined in the `iostream` header.

# Streams

- Data *usually* only flows in one direction in a stream.
- **Bidirectional** streams are streams where data flows from both endpoints in both directions.

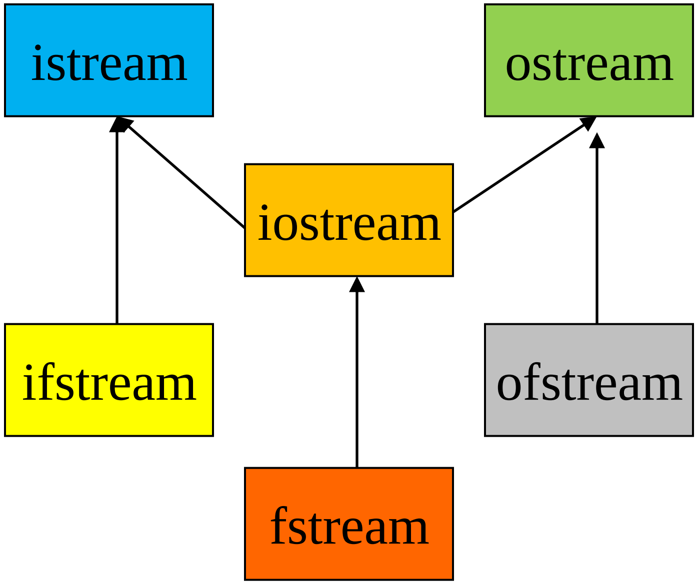
# Streams

- `cout` and `cin` are of type `ostream` and `istream` respectively. These classes form part of the C++ stream hierarchy:



# Streams

- File streams are another part of the hierarchy that enables stream input and output to files.



# Stream states

- A stream can be in one of four states
  - `good()`
  - `bad()`
  - `fail()`
  - `eof()`
- `operator!` is overloaded for streams to return true if the stream is not `good()`.

# Stream states

- `good()` – Next operation on the stream is likely to succeed.  Last operation succeeded.
- `fail()` – Last operation failed, and next operation is likely to fail, but the stream can be recovered.
- `bad()` – Last operation failed.  Next operation on the stream is likely to fail, and is probably unrecoverable.
- `eof()` – end-of-file marker was seen on the stream.

# Stream states

- A stream can be set back to the `good()` state, using the `clear()` function. Note that this does **not** fix the problem that caused the failure.

# Using streams

```
int main() {
   int value;
   cin >> value;
   if (!cin) {
      cerr << "Error! invalid input\n";
   }
   cin.ignore(1000, '\n');
   cin.clear();

   return 0;
}
```

# Using streams

- In this simple example, we have checked if we received a non-integer result (which would set `cin.fail()`), and then recovered from the error.
- We use the `ignore()` function here to skip 1000 characters, or until a new line is seen.
- We then clear the error flag.

# Stream operators

- A common practice is to define an overloaded stream output and input operator for a user defined class.
- This needs to be a free function rather than a member function of the user defined type as:
  - The left hand operand is the stream.
  - We do not have access to the stream code.

# Stream operators

- It is common to make these free functions friends of the user defined class to make accessing data members easier.
- It is also necessary to return a stream type from these functions to allow cascaded output, such as:

```
cout << “Hello” << g << endl;
```

# Stream operators

```
class Game {
 friend ostream& operator<<(ostream&, const Game&);
};

ostream& operator<<(ostream& lhs, const Game& rhs) {
   lhs << "Game score: " << rhs.points;
   return lhs;
}

//...
Game g;
cout << g << endl;
```

# Stream operators

- For an output stream operator, the first parameter and return type must be a reference to an `ostream`.
- For an input stream operator, the first parameter and return type must be a reference to an `istream`.
- Together, these functions make user defined types behave nicely with streams.

# File streams

- The file stream classes are `ifstream` and `ofstream`. These read from, and write to, files respectively.
- Being streams, these support the same ideas that we have seen from using `cin` and `cout`.
- This makes using file I/O hardly different from using screen I/O.
- These are located in the `<fstream>` header.

# Writing to a file

- First, we need to open a file:

```
std::ofstream fout(“filename.txt”);
```

- We can also provide options to open the file with.

```
std::ofstream fout(“filename.txt”,
      ios_base::trunc | ios_base::app);
```

# Writing to a file

- The following options are supported (all part of `ios_base::` class:
  - `app` – all writes go to end of file.
  - `ate` – open file, set current write position to end.
  - `binary` – open file in binary mode.
  - `in` – open file for reading
  - `out` – open file for writing
  - `trunc` – truncate file to zero length.

# Writing to a file

- To check if the file successfully opened, test if the stream is `good()`.
- Once the file is opened, we can write data to it:

```
fout << “Hello World!\n”;

fout << g << endl;
```

# Reading from a file

- Firstly, we need to open the file.

```
ifstream fin("filename.txt");
```

- The same options apply to `ifstream` as to `ofstream`, and can be specified the same way

```
ifstream fin("filename.txt", ios_base::ate);
```

# Reading from a file

- To check if the file was successfully opened, we can check if the stream is `good()`.
- We can then read from the file in the same manner as for `cin`:

```
fin >> value;
```

# Binary files

- Binary files are different to text files. If we have a text file, and we write an integer value to the file, we get the following:

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

| | | | | | |
|---|---|---|---|---|---|
| 0x40 | 0x41 | 0x42 | 0x43 | 0x44 | 0x45 |

- If we write the same integer (123456), to a binary file, we write out the binary representation to the file.

| | | | |
|---|---|---|---|
| 0x00 | 0x01 | 0xE2 | 0x40 |

0x01E240 is 123456 in hexadecimal

# Binary files

- Binary files can therefore be more efficient than text files.  However, they have hidden complexities.
- The previous example had 4 bytes written to the file with most significant byte first.  This is called **little-endian format**.
- Depending on the architecture, this might be reversed (called **big-endian**).
- Swapping files between little- and big-endian architectures will prevent the files from being read.

# Reading and Writing blocks

- One trick often used with binary files is to do the following:

```
Game g;
fout.write(static_cast<char*>(&g), sizeof(g), 1);
```

- This writes the contents of an object to the file.
- **Avoid this!** Big and little-endian dramas, and problems when you read back pointer values will guarantee to make this a problem for you.

# Reading and Writing blocks

- The way to read a block of file into memory is similar to before:

```
Game g;
fin.read(static_cast<char*>(&g), sizeof(g), 1);
```

- Only use `read()` and `write()` to read in blocks of characters or bytes. Don't try to use these for objects.

# Moving around in files

- To move around within the file, you can use the `seekp()` member function. The next read or write operation will occur from the new location.

```
// move to byte 123 in file
fout.seekp(ios_base::SEEK_SET, 123);
```

- If a file is opened with `ios_base::app`, all writes will occur at the end of the file!
- For an istream object, the member function is called `seekg()`

# Finding position in file

- To find out where the next read or write operation will occur, you can use the `tellp()` or `tellg()` functions.

```
int pos = fout.tellp();

int pot = fin.tellg();
```

# Closing a file

- To close a file, you may either explicitly call the `close()` member function, or destroy the stream object.
- The stream object's destructor automatically closes the file for you.

# Manipulating the stream

- Streams have formatting flags that control how values written to a stream are output to the endpoint.
- These flags control things like whether to output in decimal or hexadecimal, how many decimal points to display, or how wide an output field is.

# Manipulating the stream

- To set a flag, we use the `setf()` member function.
- Each flag is a constant located in the `ios_base` class.

```
// output in hexadecimal format
cout.setf(ios_base::hex);
```

# Manipulating the stream

- Here is a list of flags:
  - `left` pad field to left
  - `right` pad field to right
  - `Internal` pad between sign and value
  - `Boolalpha` display ‘true’ rather than 1
  - `dec` decimal output
  - `hex` hexadecimal output
  - `oct` octal output

# Manipulating the stream

- List of flags (cont)
  - `Scientific` floating point 1.23456E9
  - `fixed` fixed point 0.00123
  - `Showbase` place 0 or 0x in front of oct or hex numbers
  - `Showpoint` always put a decimal point and zeros
  - `showpos` always show + on positive values
  - `Uppercase` Make E in scientific uppercase

# I/O Manipulators

- Another option, rather than setting a flag, is to use an I/O manipulator.
- These are found in the `<iomanip>` header.
- All flags have corresponding manipulators.

# I/O Manipulators

- Additional manipulators are

| | |
|---|---|
| –`flush` | flush stream |
| –`endl` | put '\n' and flush |
| –`ends` | put '\0' and flush |
| –`setiosflags` | set flags via manipulator |
| –`setbase` | set output base |
| –`setfill` | set the fill character |
| –`Setprecision` | set precision of floating point |
| –`setw` | set width of field. |

# I/O Manipulators

- Try the following:

```
cout << hex << 123 << dec << 123
     << setprecision(4) << 3.14159265389 << endl;

cout << setw(4) << 1 << setw(3) << 2 << endl;

cout << setiosflags(ios_base::left) << setw(10)
     << 1 << 2 << endl;
```

# String streams

- It is also possible to set a string as the endpoint for a stream. Therefore, we can read and write from strings using the stream functionality.
- These replace `sscanf` and `sprintf` functionality from C.
- These **string streams** are located in the `<sstream>` header.

# String streams

- To write formatted output to a string, we use an `ostringstream`.

```
ostringstream ost;
ost << "Date of birth: " << day << "/"
    << month << "/" << year;
```

- To access the underlying string, use the `str()` member function.

```
string s = ost.str();
```

# String streams

- Retrieving data from a string is done using an `istringstream` object.

```
string s = "11 22 33";
istringstream ist(s);
int a, b, c;
ist >> a; // contains 11
ist >> b; // contains 22
ist >> c; // contains 33
```