CSCI851 Spring-2021 Advanced Programming Lecture 04

C++ Foundations VI: Handling files

Outline

Text file streams.

- Errors in opening files.
- Errors in reading files.

Character input.

Buffering.

Binary I/O.

Text File Streams

```
#include <fstream>
using namespace std;

int main()
{
    ifstream inData;                    //declare an input file stream
    ofstream outData;                   //declare an output file stream
    string firstName, lastName;

    inData.open("names.txt");           // open input file
    outData.open("marks.txt");          // open output file

    inData >> firstName >> lastName;    // read from a file stream
    outData << 85.6;                    // write into a file stream

    inData.close();                     // close the input file
    outData.close();                    // close the output file
    return 0;
}
```

Unbounded file streams …

On the previous slide we opened an input stream, and we opened an output stream.

We could have used `fstream`, an unbounded file stream allowing both reading and writing.

```
fstream fstrm;
fstrm.open(s, mode);
```

Modes, for this and the i-o versions, partially:

```
in, out, app, ate, trunc, binary
```

Multiple modes can be set.

They are part of `fstream::` …

`app`: Always write at the end.

`ate`: On opening go the end.

`trunc`: Wipes out existing content.

`binary`: Doesn't have the text conversion layer, you read/write in binary.

There are constraints on these flags.

Some examples:

- `out` is for `fstream` or `ofstream`.
- `trunc` can only be set if `out` is.
- `app` and `trunc` are mutually exclusive.

Errors in opening files …

Don’t assume a file stream has been opened successfully.

- Incorrect file name:

```
inFile.open("names.tx1");
```

- Incorrect file opening mode:

```
ifstream inFile;
inFile.open("names.txt", ios::trunc);
```

- Not enough room on the hard drive.
- Hardware failure.

Always check the status of a stream after `open`.

```
#include <fstream>
using namespace std;

int main()
{
    ifstream inData;        // declare an input file stream
    char fileName[] = "exams.txt";
    string lastName, mark;

    inData.open( fileName );        // open input file
    if (!inData)            // check if opened successfully
    {
      cerr << "Error opening : " << filename << endl;
      return -1;            // exit with an error code
    }
    inData >> lastName >> mark;

    inData.close();         // Close the input file
    return 0;
}
```

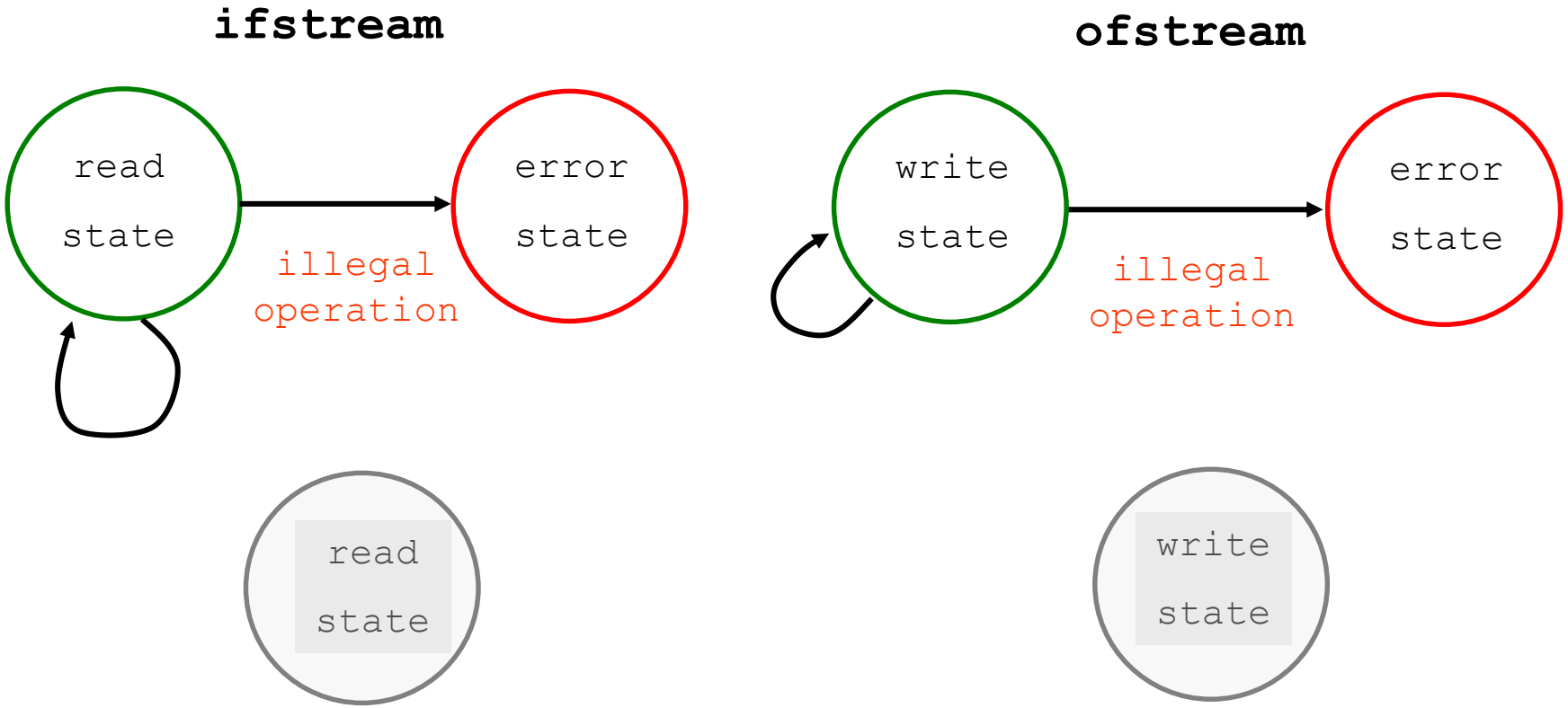
Errors in reading files …

So the file seemed to open okay but …

… being pessimistic, what goes wrong next.

- The program may not have data to read as it hits the end of file.
- The data may be invalid: … an alphabetic character instead of a digit character; a control character instead of an alphabetic one; etc.
- The data may not be physically accessed from the disk due to its damage or network failure.

Error States



Any illegal operation with a stream switches it to the Error State !

Comprehensive error checking is needed, and appropriate error recovery.

- C++ provides three status flags and four functions to detect possible errors.

1. The flag `eof` indicates that the end of file is reached.

```
if( inData.eof() )  {  Error recovery action }
```

2. The flag `fail` indicates a failure due to invalid data.

```
if( inData.fail() ) {  Error recovery action }
```

3. The flag `bad` indicates a hardware problem.

```
if( inData.bad() ) {  Error recovery action }
```

4. The function `good()` returns true if no any error has been detected.

Note that files are viewed here as sequences of bytes with an end-of-file character at the end.

What happens?

A text file has content:

```
1 2 3
```

Assuming appropriate headers etc …

```
while( ! inData.eof() )
  {
      inData >> number;
      cout << number << " ";
  }
```

Produces output …

```
1 2 3 3
```

Caution: `eof()` doesn't test for the end-of-file. It simply returns a value of the corresponding indicator. The indicator is changed by >> attempts.

Error recovery …

Once the stream is in the error state, it will stay that way until you take specific action:

- All subsequent operations will do nothing, or loop forever no matter what they are or what is in the input.
- You have to clear the stream by calling `clear()` to recover the stream from the fail state.

```
inFile >> newNumber;
if( inFile.fail() )
{
    inFile.clear();
    inFile.ignore(100, '\n');
}
```

Recovers the file stream from the error state. However, further reading of data may be useless as the wrong characters are still in the stream buffer.

Discard 100 characters (or until the end-of-line indicator) from the stream buffer.

Example (fixed format text file)

```
int readData(ifstream& inFile, InfoType& student, string myId)
{
   string nameFirst;
   string nameLast;
   string Id;

   do inFile >> nameFirst >> nameLast >> Id;
   while( inFile.good() && Id != myId );

   if(inFile.fail())  return -1;          // invalid character
   if(inFile.bad())   return -2;          // hardware failure
   if(inFile.eof() && Id=="") return -3; // myId not found

   student.firstName = nameFirst;
   student.lastName = nameLast;
   student.Id = Id;

   return 0;
}
```

Character input …

How do we read **all** characters from the input stream; including blanks, tabs, and new-lines?

- This could be a input file stream or something else, like standard in.
- The extraction operator doesn't read white space or characters.

You can use `get` functions to read a character:

```
ifstream inFile;
char nextChar;
nextChar = inFile.get();
```

You can use `getline` to read a line of characters from a text file.

```
char lineBuffer[bufSize];
infile.getline( lineBuffer, bufSize );
```

Be careful.

test.txt

```
34.99
Motor Oil
```

```
float price;
char productName[20] ;
char fileName[] = "test.txt";
ifstream inData;

inData.open( fileName );

inData >> price;

inData.getline(productName, 20);

cout << price << endl;
cout << productName << endl;
```

The newline character is still in the stream buffer, and that is interpreted as an empty string.

So, we need to clear the buffer …

```
inData >> price;
inData.ignore( 20, '\n' );
inData.getline(productName, 20);
```

A different form and a delimiter

It's often unreasonable to forecast the input length.

So we can use this form …

```
getline(cin,input);
```

This has a default delimiter or endpoint of \n, that's a new line.

But you can change this …

```
getline(cin,input,'t');
```

Be careful with this, you can capture a lot more than you expect.

Character output
Output formatting

Use `put()` in a similar way to `get()`.

You can use output manipulators if you want to format your output:

```
cout << setiosflags(ios::fixed);
cout << setiosflags(ios::showpoint);
cout << setprecision(2);
```

Common Escape Sequences

Table 2-4 Commonly Used Escape Sequences

	Escape Sequence	Description
\n	Newline	Cursor moves to the beginning of the next line
\t	Tab	Cursor moves to the next tab stop
\ b	Backspace	Cursor moves one space to the left
\r	Return	Cursor moves to the beginning of the current line (not the next line)
\\	Backslash	Backslash is printed
\'	Single quotation	Single quotation mark is printed
\"	Double quotation	Double quotation mark is printed

Buffering

When you direct data into a file, it is not sent there immediately.

- It is physically written to the device only when the buffer is full.

When you read data from a file, you input it from the input buffer.

- When the buffer becomes empty it is refilled with a new block of data.

Why buffer?

- It reduces the number of accesses to the external device.

Only `cerr`, standard error, is never buffered.

- Errors are critical.

Text and Binary Files

Text files:

- composed of characters
- data types have to be formatted/converted into a sequence of characters
- variable number of characters (bytes) for the same data types
- usually sequential access

Binary files

- binary numbers (bits)
- fixed size of the same data types
- random access
- more compact, less portable than text files

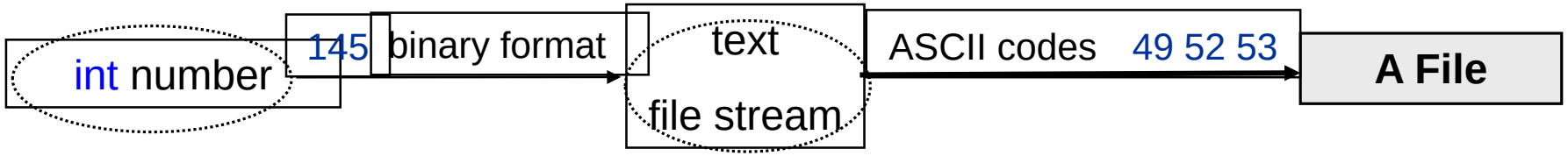
Text Files Input/Output

Text File Input/Output functions also carry out data type conversion:

Example:

```
int number;
outFile << number ;
```

The **<<** operator converts `int` into a sequence of ASCII codes and writes them into a file.

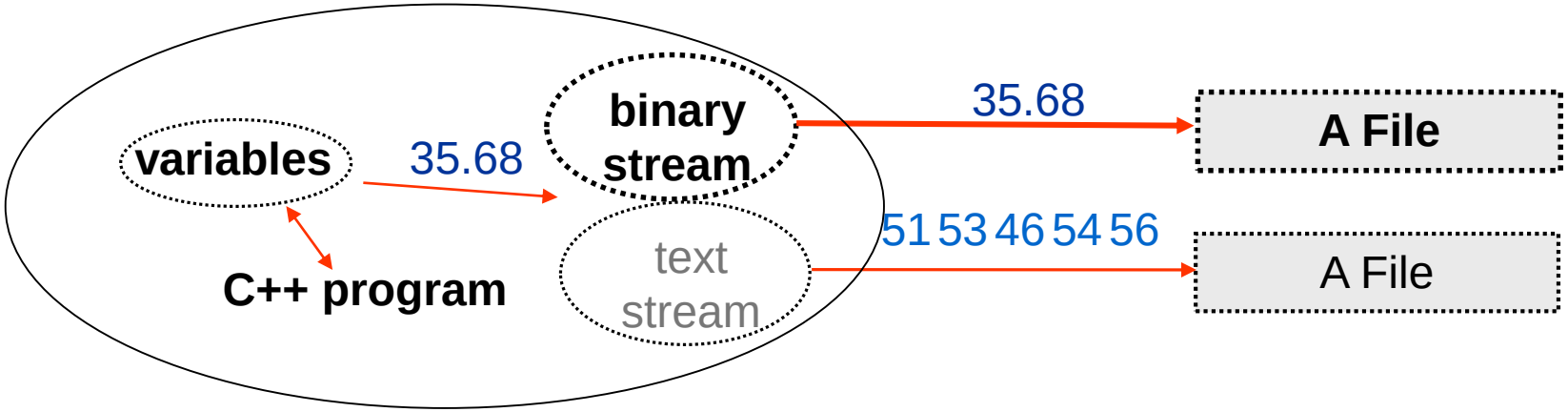



```
inFile >> &number;
```

The **>>** operator converts a sequence of ASCII characters into an integer number and stores it into a variable.

Binary File Stream Concept

A Binary File Stream is an interface between a program and a physical file that **does not perform any type conversion .**



File Streams can be Text or Binary, but physical files do not have any special marker to indicate their type.

A file name or its extension does not affect the file type.

Getting organised I:
Structs, unions, and randomness

Outline

Abstract data types …

- Structs.
- Unions.

Randomness…

Abstract data types

It’s quite common to have related elements of data, particularly when we are modelling some non-trivial entity.

- Humans, for example, have a lot of characteristics, not just a name.

Once we start dealing with multiple instances of something, we want to be able to distinguish between the characteristics, and distinguish between them for the instance they are associated with.

For example, let's say that we are writing a program related to a cat.

- An individual cat might be described by a name, mass, tail length, and colouring.

If there is only one cat we might reasonably and unambiguously use

```
string name;
float mass;
float tailLength;
string colouring;
```

But what happens when we have lots of cats ...

We could use arrays …

```
const int numberCats = 5;
string name[numberCats];
float mass[numberCats];
float tailLength[numberCats];
string colouring[numberCats];
```

… but it's tidier to have some way of connecting the data elements other than simply through the index.

- If there is ever any reason to change an index value, sorting cats for example, we need to make sure the indices line up.

So, we define our own abstract data types, that typically contain multiple related data elements.

Abstract data types …

We will typically use classes to provide this encapsulation, but there are a couple of related pre-class entities:

- The `struct`:
  - This is a special kind of class so it's not overly exciting.
- The `union`:
  - This is special kind of class too, but a more interesting special.

The `struct` construct

A C++ struct is a way to group related data elements together, in a similar way to a C++ or Java class.

- **Aside for now**: A struct differs only from a class in that the access specifiers default to public whereas in classes they default to private.
- Typical programming style has structs only being used in C++, in place of classes, if all the members are going to be public.
- Everything being public breaches encapsulation so there should be a good reason for doing this…

A struct declaration ends with a semi-colon (;).

```
struct Cat {
   string name;
   float mass;
   float tailLength;
   string colouring;
};
```

```
struct Student {
   string name;
   int id;
};
```

The structure name `Student` is a new type, so you can declare variables of that type, for example:

```
Student s1, s2;
```

To access the individual fields of a structure, we use the dot operator:

```
s1.id = 123;
s2.id = s1.id + 1;
s1 = s2;   // copies fields of s2 to s1
```

It's possible to have instances of structs inside structs…

```
struct Address {
  string city;
  int postCode;
};

struct Student {
  string name;
  int id;
  Address addr;
};
```

To access nested structure fields use more dots…

```
Student s;
s.addr.postCode = 2500;
```

Can structs have member functions?

Linking data elements seems sensible but if we think about functions that only act on data within a struct, having linked functions seems reasonable too.

- And sure, structs can have functions too.
  - Remember they differ from classes only in the default access specifiers.

Structs are often used to declare simple datatypes, but don't have to be.

The problem with putting member functions in Structs is that unless you remember to add in access specifiers people don't have to use the interface you provide, the data is public by default ☹

So, we would expect to use classes when we want to control the way people interact with the data.

```
struct Test {
    string name;
    int number;

    void setTest(string, int);
    void showTest();
};

void Test::setTest(string TestName, int TestNumber) {
    name = TestName;
    number = TestNumber;
}

void Test::showTest() {
  cout<<"Test string " << name << endl;
  cout<<"Number for this  "<< number << endl;
}
```

```
int main()
{
      Test myTest;
      myTest.setTest("Bob", 19);
      myTest.showTest();
}
```

```
int main()
{
      Test myTest;
      myTest.name=Bobby;
      myTest.number=15;
      myTest.showTest();
}
```

Static consts in structs …

If you have a static const you can only declare and initialise it in a struct, or class, if it's of integral or enumeration type.

– Or if it's initialised by a constant expression.

Otherwise you have to initialise it outside…

```
struct Trial {
   static const double trial;
};
const double Trial::trial = 11.73;
```

Static consts in structs …

From Stroustrup: http://www.stroustrup.com/bs_faq2.html

"So why do these inconvenient restrictions exist? A class is typically declared in a header file and a header file is typically included into many translation units. However, to avoid complicated linker rules, C++ requires that every object has a unique definition. That rule would be broken if C++ allowed in-class definition of entities that needed to be stored in memory as objects."

The following is okay:

```
static constexpr double trial=11.73;
```

Another type of type: The `union`

A union declaration is similar to a `struct` declaration, but the fields of a `union` all share the same memory.

```
union mytype {
      int i;
      float f;
};
```

So assigning values to fields: ‘i’ or ‘f’ would write to the same memory location.

You should use a union when you want a variable to be able to have values of different types, but not simultaneously.

Costing a construct …

Structs and unions are special cases of class, and while we can return to this later, it's useful to note their sizes.

A struct is just the concatenation of the data members, not so with the union.

```
struct adt {
   int i;
   float f;
};
```

```
union mytype {
      int i;
      float f;
};
```

```
cout << sizeof(int) << " " << sizeof(float) << endl;
cout << sizeof(adt) << endl;
cout << sizeof(mytype) << endl;
```

```
4 4
8
4
```

Randomness: Old school…

Pre C++11 C++, and C, relied on the use of a C library function `rand`…

- Uniform distribution in the range 0 to X.
  - With X a system dependent value, but at least 32767…

```
#include <iostream>
using namespace std;

int main()
{
        srand(time(0));

        for ( int i=0; i < 20; i++)
                cout << rand() << endl;

        return 0;
}
```

Seeding the random generator with time.

Randomness? Seeding?

When talking about generating randomness in programming we are typically meaning generating sequences of numbers using PSRG's: pseudorandom numbers generators.

- It's pseudorandom because it is likely to look random and have difficult to find patterns, but the patterns will be there, it's deterministic.
- Same input, same output.
- Similar input, expect very different sequence.

Seeding is setting a start point/initial point for our sequence.

Limitations:

- You often want a different range.
- You often don't want an integer.
- You often don't want a uniform distribution.
  - You want some values to appear more than others.
- Even with the same seed, you aren't guaranteed to get the same sequences on different platform.
  - This is a problem if you are using generator and seed as a way of storing something like a game level.
- There are also global state problems that make testing difficult since the state of `rand()` can be modified by calls from anywhere.

So classically …

You would get integers in the range `rand` gives you and try to map them to whatever you need.

- This can often be done in a way that gives a non-uniform distribution, even though a uniform distribution may be what you want.

You could use assertions to test if the randomness matches what you expect, but there can be problems doing that because of calls from elsewhere if you have multiple threads.

Randomness: C++11

Now you should use the random-number engine, with the library `random` included.

```
$ CC -std=c++11 rand-eng.cpp
```

```
default_random_engine randEng;
for ( int i = 0; i < 10; i++)
    cout << randEng() << endl;
```

This is default seeded … the output is fixed.

Before we used the default constructor, but we can use one with an integral seed too.

```
default_random_engine randEng(seed);
```

Or set the seed later..

```
randEng.seed(seed);
```

And we can see the ranges simply using …

```
randEng.min()        randEng.max();
```

Distributions and ranges …

Uniform between in some range…

```
uniform_int_distribution<unsigned> uniform(0,9);
default_random_engine randEng;
for ( int i = 0; i < 10; i++)
  cout << uniform(randEng) << endl;
```

There are other distributions you might want to use, such as the normal distribution, where the constructor takes a mean and standard deviation:

```
normal_distribution<> normal(5,1.5);
```

Integers with a normal distribution

The textbook shows how you can use the `lround` function to round the values obtained to the nearest integer.

```
for ( int i = 0; i < 10; i++)
  cout << lround(normal(randEng)) << endl;
```

Some notes …

Make sure the `randEng` construction isn't in a loop.

If you have a local generator function, which sets up the range and distribution type, you should declare both the engine and distribution as being static…

```
static default_random_engine randEng;
static uniform_int_distribution<unsigned> uniform(0,9);
```

- You may want different ranges …

Be careful using `time(0)` as a seed.

- It's predictable.
  - This may be a good thing, but typically if the randomness is to do with security you don't want predictability.
- It's the same if you are running the same process at roughly the same time.

Mersenne Twister: `std::mt19937`

Period before repeating : $2^{19937}-1$.

```
std::mt19937 mtg;
```

The `mt19937` is a convenient typedef.

Range: 0 to 4294967295.

```
mtg.min()        mtg.max();
```

It's not a cryptographically secure PRNG.

Here goes an example of a function for returning a random integer in a specified range.

```
int randint (int x) {
   static std::mt19937 mtg(time(0));
   return std::uniform_int_distribution<int> (0, x-1) (mtg); }
```

The generator is static and you are returning a value from the appropriate range.

```
for (int i=0;i< 10;i++)
   std::cout << randint(10+10*i) << std::endl;
```

Getting organised II:
A little bit of Time

Outline

OS time on Banshee.

C-style: std::time

C++11: std::chrono

Looking at the time …

```
$ time ./calling

real    0m0.046s
user    0m0.032s
sys     0m0.009s
```

If we want to know how long our program takes overall, on Banshee we can use `time`.

Real: Start to end.

User: CPU time in user mode.

Sys: CPU time in sys mode.

Total CPU time: user+sys

If you were using a multi-processor system you can have Real < user+sys.

C-style dates and times
std::time in header <ctime>

See https://en.cppreference.com/w/cpp/chrono/c/time

```
#include <ctime>
#include <iostream>

int main()
{
      std::time_t result = std::time(nullptr);
      std::cout << std::asctime(std::localtime(&result))
            << result << " seconds since the Epoch\n";

      std::cout << CLOCKS_PER_SEC << std::endl;
}
```

Time in C++11

http://en.cppreference.com/w/cpp/chrono

Three main types are defined:

- Clocks:
  - With an epoch and a tick rate.
    - The tick rate isn't necessarily
- Time points:
  - Duration of time since epoch.
- Durations:
  - Span of time associated with some number of ticks.

```
#include <iostream>
#include <chrono>

long fibonacci(unsigned n)
{
    if (n < 2) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main()
{
    auto start = std::chrono::system_clock::now();
    std::cout << "f(35) = " << fibonacci(35) << '\n';
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "finished computation at " << std::ctime(&end_time)
              << "elapsed time: " << elapsed_seconds.count() << "s\n";
}
```

Sleeping …

https://en.cppreference.com/w/cpp/thread/sleep_for

http://www.cplusplus.com/reference/chrono/

With the headers `thread` and `chrono`, you can access functionality to have your program wait for some specified length of time.

```
std::this_thread::sleep_for(std::chrono::seconds(2));
std::this_thread::sleep_for(std::chrono::milliseconds(2));
```

Time durations …

```
Hours, minutes, seconds, milliseconds,
  microseconds, nanoseconds.
```

Why? → Monitoring across multiple threads.

PLEASE: Don’t use sleep in the assignments.

C++20

There is a lot of support proposed for managing calendars and time.

There are some specific Clocks, associated with various times.

And functionality for converting between clocks and for partitioning time.

Getting organised III: Debugging and profiling: dbx and gprof

Debugging … with dbx

The syntax for this is non-examinable.

In some of the labs you have used the compiler to pick up problems through compilation time errors and warnings.

That's useful but having a program compile is usually only part of the battle, we should expect there to be run time problems too.

A debugger is used to step through our programs as they run, and help us pick up errors in our programs.

Knowing how to use a debugger will likely help at some point…

The debugger `dbx` is available on Banshee.

- There is an X11 graphical version, `dbxtool`.
- It works for Java too.

Documentation at:

https://docs.oracle.com/cd/E24457_01/html/E21993/

Using dbx

You need to compile a program with –g flag.

– This pretty much just stores ties to the original.

So for Test.cpp,

```
$ CC -g Test.cpp -o test
```

We are going to initially use C++ code that doesn't clear dynamic memory correctly to see how the debugger can pick up leaks with a bit more information than `bcheck`.

Test.cpp

```
#include<iostream>
using namespace std;

int main()
{

        int *p;
        p = new int(5);
        cout << p << endl;

        return 0;
}
```

Having compiled our test programs with the debugger information turned on we can run `dbx` for debugging.

```
$ dbx test
```

This loads the program into the debugger, ready for working on.

To run …

```
(dbx) run
```

If you run dbx without the command line argument you can use …

```
(dbx) debug test
```

Memory leaks …

```
(dbx) check -memuse
memuse checking - ON
(dbx) run
Running: test
(process id 18558)
Reading rtcapihook.so
Reading libdl.so.1
….
Reading libmd_psr.so.1
RTC: Enabling Error Checking...
RTC: Running program...
276d8
Checking for memory leaks...

Actual leaks report    (actual leaks:            1  total size:          4
bytes)

  Total     Num of  Leaked     Allocation call stack
  Size      Blocks  Block
                    Address
==========  ======  ==========  ======================================
         4       1    0x276d8  operator new < main
```

```
struct Test {
    string name;
    int number;

    void setTest(string, int);
    void showTest();
};

void Test::setTest(string TestName, int TestNumber) {
    name = TestName;
    number = TestNumber;
}


void Test::showTest() {
  cout<<"Test string " << name << endl;
  cout<<"Number for this  "<< number << endl;
}
```

```
int main()
{
      Test myTest;
      myTest.setTest("Bob", 19);
      myTest.showTest();
}
```

```
int main()
{
      Test myTest;
      myTest.name="Bobby";
      myTest.number=15;
      myTest.showTest();
}
```

Test2.cpp

We can get the debugger to step through our program, for example stopping when we get to a particular function…

```
(dbx) stop in showTest
(2) stop in Test::showTest()
(dbx) run
Running: test
(process id 20547)
Reading libc_psr.so.1
stopped in Test::showTest at line 19 in file "Test2.cpp"
   19     cout<<"Test string " << name << endl;
(dbx) where
=>[1] Test::showTest(this = 0xffbff78c), line 19 in "Test2.cpp"
  [2] main(), line 28 in "Test2.cpp"
(dbx)
```

Or give a code line.

At those breakpoints we can ask for variable values using `print`, as in the following example for our struct program…

```
(dbx) print name
name = "Bobby"
(dbx) print name[0]
More than one identifier 'operator[]'.
Select one of the following:
 0) Cancel
 1) std::basic_string<char,std::char_traits<char>,std::allocator<char> >::operator[](unsigned)const
 2) std::basic_string<char,std::char_traits<char>,std::allocator<char> >::operator[](unsigned)
> 1
name.operator[](0) = 'B'
(dbx) print &name
&name = 0xffbff70c
```

Profiling …

Another tool that can be used to help our coding is a profiler.

On Banshee we have `prof` and `gprof`.

We will briefly explore this in the context of our `Test2.cpp` struct test program.

For compilation we now use the flag –p if we intend to use prof …

```
$ CC -p Test2.cpp -o test
```

and –gp to prepare for using gprof.

```
$ CC -gp Test2.cpp -o test
```

When you do the compilation you will get a significantly larger executable than you would without the –p argument, the –g only adds a little.

When you run that larger executable …

```
$ ./test
```

… you get an output file `mon.out`, or `gmon.out`, for use with `prof` and `gprof` respectively.

Then, then depending on the –g flag, run

```
$ prof test
```

or

```
$ gprof test
```

The Banshee man pages …

You can use the `man` pages on Banshee to get information on utilities.

– It's short for manual…

Use it as …

```
$ man command
```

So …

`prof`: display profile data

`gprof`: display call-graph profile data

… `gprof` provides more detailed information.

calling.cpp

```
#include <iostream>
using namespace std;

void funA(){for (int x=0;x<100;x++){}};
void funB(){for (int x=0;x<1000;x++){}};
void funC(){for (int x=0;x<10000;x++){}};

int main()
{
        srand(time(0));
        for (int x=0; x< 1000; x++)
        {
                switch ( rand() % 3 ){
                        case 0: funA();
                                break;
                        case 1: funB();
                                break;
                        case 2: funC();
                                break;
                        default:
                                break;
                }
        }
        return 0;
}
```

Why?
Functions with different costs…

Let's look at the output from `prof` first …

```
$ CC -p calling.cpp -o calling
$ ./calling; prof calling
```

The semi-colon is used to chain commands.

– Here it makes it easier to repeat both together.

```
%Time Seconds Cumsecs  #Calls   msec/call  Name
100.0    0.03    0.03     325      0.09   __1cEfunC6F_v_
  0.0    0.00    0.03     329      0.00   __1cEfunA6F_v_
  0.0    0.00    0.03     346      0.00   __1cEfunB6F_v_
  0.0    0.00    0.03       1      0.     main
  0.0    0.00    0.03       1      0.     __1cH__CimplKcplus_fini6F_v_
  0.0    0.00    0.03       1      0.     __1cG__CrunSregister_exit_code6FpG_v_v_
  0.0    0.00    0.03       1      0.     _ex_deregister
  0.0    0.00    0.03       1      0.     __1cH__CimplQ__type_info_hash2t5B6M_v_, __1cH__CimplQ__type_info_hash2t6M_v_
```

Back to calling `gprof`

Similar to before …

```
$ CC -pg calling.cpp -o calling
$ ./calling; gprof calling
```

The output is a fair bit longer.

The output from `prof` is a subset of the output from `gprof`.

We aren't going to make any attempt to understand this output.