Macros

- Allow programmer to tell compiler how to rewrite source code just prior to actual compilation
- Lets programmer use syntax in the source code that isn't a built-in part of the language
- Support for macros, how much flexibility/control programmer has for re-writes, varies widely across languages, e.g.
    - C: #defines (preprocessor directives)
    - C++ templates
    - Lisp macros (seen before study break)

C preprocessor directives

- #defines for simple text substitution, e.g.

```
#define Pi 3.14

X = Pi; // replaced with X = 3.14; by preprocessor
```

- allowed C to give constant-like behaviour when language did not directly support constants (programmer #defined to associate name with value, used name in rest of program)
- not the same as constants: not scoped, and actually performs text substitution prior to compilation

Built-in #defines

- A number of pre-defined #defines, handy for debugging
- `__FILE__` name of current file
- `__LINE__` number of current line in file
- `__DATE__` current date
- `__TIME__` current time
- plus many others...

#undef

- Can “undefine” a previously defined value (possibly to redefine with new value afterward)

```
#define Pi 3.14
....
#undef Pi
#define Pi 3.1415
```

C pre-processor (cont.)

- Conditional code inclusions at compile time using #ifdef, #ifndef (e.g. “guards” in header files), can also be used to include/exclude groups of code at compile time, e.g.

```
#ifdef __unix__
   Code to use when compiled on linux
#elif defined _WIN32
   Code to use when compiled on windows
#endif
```

C preprocessor macros

- Parameterized #defines to give appearance of functions

```
#define swap(a,b)  { int tmp = a; a = b; b = tmp; }
swap(x,y); // replaced with
           // { int tmp = x; x = y; y = tmp; }
```

- Potential name clashes, type mismatches, e.g.

```
float tmp;
int t;
swap(tmp, t); // preprocessing replaces with
              // { int tmp = tmp; tmp = t; t = tmp; }
```

Line continuation

- Can create multi-line macros by ending each line (except the last one) with a \

```
#define intswap(x,y)   { \
    int tmp = x; \
    x = y; \
    y = tmp; \
}
```

Text substitution expands possibilities

- Can do some things with preprocessor macro you couldn't with actual function call, e.g. "pass" types

```
#define swap(t,x,y) { t tmp = x; x = y; y = tmp; }
swap(float, a, b);  // becomes
    // { float tmp = a; a = b; b = tmp; }
```

- ## operation can be used to concatenate text, e.g.

```
#define swap(t,x,y) { t x##y = x; x = y; y = x##y; }
swap(float, foo, blah); // becomes
// { float fooblah = foo; foo = blah; blah = fooblah; }
```

C++ templates

- Common use is to create generic functions/classes
  (programmer creates skeletal version using identifiers instead of data types, compiler identifies correct types to use and creates/fills in versions of the function/class that are actually needed, based on rest of source code)
- Can also be used to create variadic functions (next session)
- Can also be used to create functions that run at compile time to compute content to be used during compilation

C++ template programming

- Assuming you’re used to “normal” use of templates
- Will cover templates for variadic functions shortly
- Here we’ll look at templated functions that actually execute during preprocessing, and can return constants that can be used in rest of compilation process\
- The compiletime functions can only take constants (or constant expressions) as parameters, can only return constants (or constant expressions)
- New type used: constexpr

Compile time computation example

- e.g. create a function that runs at compile time, takes a (constant) struct and a (constant) int as parameters and returns a (constant) fraction

```
struct Fraction { int n; int d; }
const Fraction f = { 11, 3 };
constexpr Fraction scale(const Fraction f, const int s) {
    const Fraction r = { f.n*s, f.d };
    return r;
}
const Fraction x = scale(f, 3); // runs at compile time
```

Uses of compile time processing

- this is actually turing complete, so could theoretically write entire programs that executed during compilation
- in practice, intent is to allow automated processing of constant data sets/values
- e.g. you #include some .h file with lots of data of some form, then use your compile-time functions to compute relevant metadata about it ... let the compiler do the work