Compilation

Objectives

- Introduce
  - preprocessor
  - separate compilation
- Examine issues of code organization

Preprocessing

- *Preprocessing* is the first stage of source-code translation
  - performed before compilation
  - provides many useful services: #include, #define, #if, etc.

Conditional compilation

- Code can be conditionally compiled
  - use #if / #endif directives

define →
test →

```
#define VERSION 201

...

#if VERSION > 200
void loadAdvancedMenus()
{
  ...
}
#endif
```

Defined symbols

- Symbol can be defined with no value
  - can test if symbol defined

define →

test →

```
#define DEBUG

...

void process()
{
#ifdef DEBUG
  cerr << "processing";
#endif

  ...
}
```

Preprocessor vs. code

- Can achieve conditional code using either the preprocessor or a variable: each technique has advantages

```
#define DEBUG

void process()
{
#ifdef DEBUG
  cerr << "processing...";
#endif

  ...
}
```

preprocessor removes code if not active to get a smaller executable

```
bool debug = true;

void process()
{
  if (debug)
  {
    cerr << "processing...";
  }
  ...
}
```

variable can be set dynamically to turn on/off at runtime

Macros

- #define can be used to create a macro, the preprocessor performs symbolic expansion (i.e. symbol replaced by text)

define →

use →

```
#define TRACE writeTrace(__FILE__, __LINE__)

void writeTrace(const char* file, int line)
{
  cerr << "At line " << line << " of file " << file << endl;
}

void process()
{
  TRACE;
  ...
}
```

Parameters

- #define macros may be parameterized, the preprocessor will perform textual substitution

```
#define TRACE(msg) writeTrace(msg, __FILE__, __LINE__)

void writeTrace(const char* m, const char* file, int line)
{
  cerr << file << "," << line << ": " << m << endl;
}

void process()
{
  TRACE("Beginning of process");
  ...
}
```

Code generation

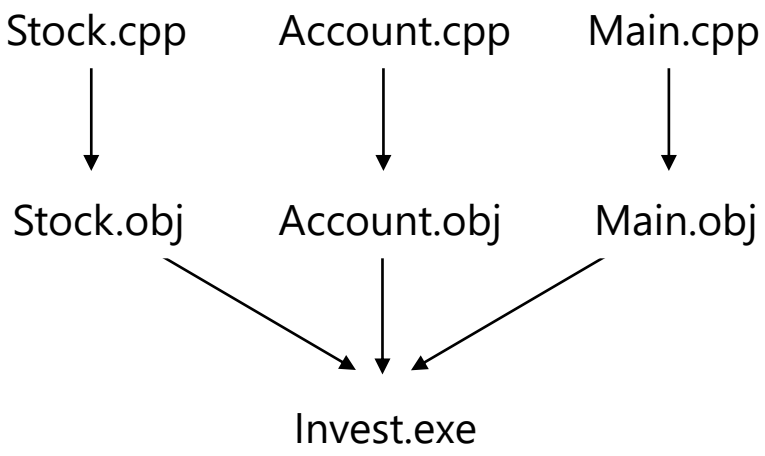
- #define macros can be used to generate code from pattern
  - line continuation with \
  - token pasting with ##

```
#define MEMBER(type, name) \
 public: \
  type name() const { return m_##name; } \
  void set##name(type v) { m_##name = v; } \
 private: \
  type m_##name;
```

```
class Position
{
  MEMBER(string, name)
  MEMBER(double, shares)
  MEMBER(double, price)
  ...
};
```

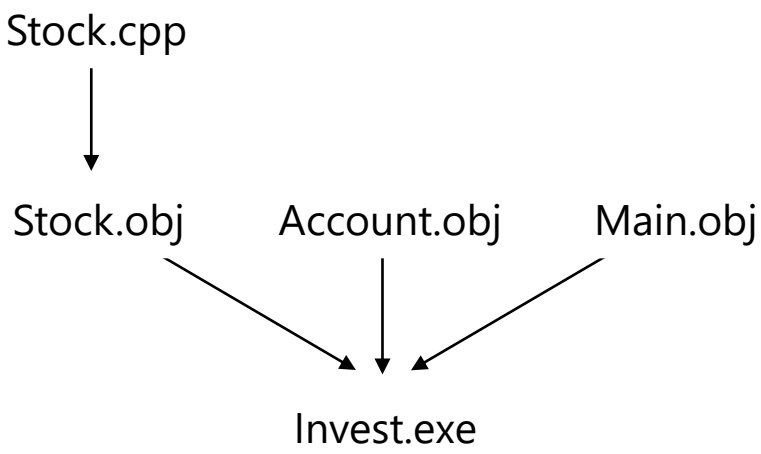
Compilation

- Code typically divided into many files
  - related code put into same file
- Source files compiled to get object files
  - object files linked to produce executable



Separate compilation

- Files can be compiled separately
  - only recompile modified files



Cross references

- Code may refer to functions in other files, client uses prototypes so the compiler will allow the call

prototype →

Main.cpp

```
void process();

int main()
{
  ...

  process();
}
```

Util.cpp

```
void process()
{
  ...
}
```

Global variables

- Use extern to refer to a global variable in another file

extern →

Main.cpp

```
extern int version;

void process();

int main()
{
  ...

  if (version > 105)
  {
    process();
  }
}
```

Util.cpp

```
int version;

void process()
{
  ...
}
```

Header files

- Header files can contain common declarations

include →

Main.cpp

```
#include "Util.h"

int main()
{
  ...
  if (version > 105)
  {
    process();
  }
}
```

Util.h

```
extern int version;

void process();
```

Util.cpp

```
#include "Util.h"

int version;

void process()
{
  ...
}
```

Class organization

- Classes often organized into header/source file pairs
  - include class header files where needed

Stock.h

```
class Stock
{ ...
  void buy();
  void sell();
};
```

Main.cpp

```
#include "Stock.h"

int main()
{
  Stock ibm;
  ...
}
```

Stock.cpp

```
#include "Stock.h"

void Stock::buy()
{
  ...
}

void Stock::sell()
{
  ...
}
```

Local include files

- Double quotes add current folder to search
  - standard search path checked if file not found

use double quotes →

Main.cpp

```
#include "Stock.h"

int main()
{
  Stock s;

  ...
}
```

Dependencies

- Header files often have dependencies on other headers

Point included indirectly →

Main.cpp

```
#include "Circle.h"

int main()
{
  Circle c;

  ...
}
```

Point.h

```
class Point
{ ...
  int x;
  int y;
};
```

dependency →

Circle.h

```
#include "Point.h"

class Circle
{ ...
  Point center;
  int   radius;
};
```

Multiple inclusion

- Multiple inclusion may cause errors
  - defining a class twice, for example

Main.cpp

error, **Point** multiply defined →

```
#include "Point.h"
#include "Circle.h"

int main()
{
  Point  p;
  Circle c;

  ...
}
```

Point.h

```
class Point
{ ...
  int x;
  int y;
};
```

Circle.h

dependency →

```
#include "Point.h"

class Circle
{ ...
  Point center;
  int   radius;
};
```

Guarding header files

- Each header given unique symbol
  - undefined in first pass
  - defined in subsequent passes

Main.cpp

Point included once

```
#include "Point.h"
#include "Circle.h"

int main()
{
...
}
```

Point.h

guard

```
#ifndef POINT_H
#define POINT_H

class Point
{
  int x;
  int y;
  ...
};

#endif // POINT_H
```

Namespace partition

- Namespaces may be in parts
  - can spread contents across multiple files

Stock.h

```
namespace Finance
{

  class Stock
  {
    ...
  };

}
```

Account.h

```
namespace Finance
{

  class Account
  {
    ...
  };

}
```

Client.h

```
namespace Finance
{

  class Client
  {
    ...
  };

}
```

Namespace headers and source

- Can separate namespace code into headers and source
  - implementations must be inside namespace

Stock.h

```
#ifndef FINANCE_STOCK_H
#define FINANCE_STOCK_H

namespace Finance
{

  class Stock
  {
    ...
  };

}

#endif // FINANCE_STOCK_H
```

Stock.cpp

```
#include "Stock.h"

namespace Finance
{

  void Stock::buy()
  {
    ...
  }

  void Stock::sell()
  {
    ...
  }
}
```

Hierarchical organization

- Can use subfolders for entire libraries
  - common to mirror namespaces

```
#include "Finance/Stock.h"
#include "Finance/Account.h"
#include "Db/History.h"
#include "Db/Record.h"
...

int main()
{
  ...
}
```

Summary

- Preprocessor commands modify source before compilation
  - remove/select code
  - generate code
- Code may be partitioned into multiple files
- Include files used to hold definitions
  - classes
  - function prototypes
  - extern
- Guards prevent multiple inclusion