An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 25

The Last Lecture

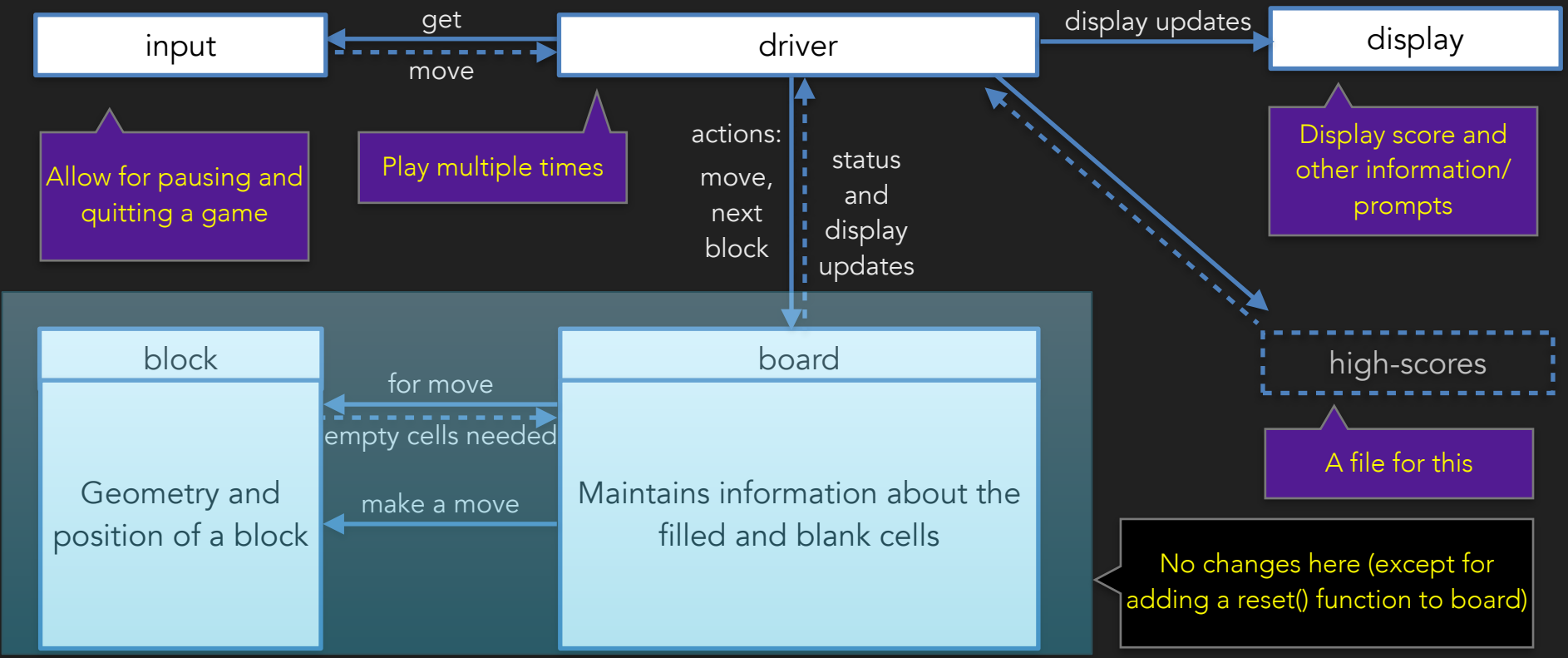
Bells & Whistles, and Concluding Remarks

Based on material developed by Prof. Abhiram G. Ranade

Today

- Recall the Tetris game from last time
- Will add some extra features today
- A few more C++ features along the way (not all used in our program)
  - union, switch statement, std::fill, std::getline, std::time_t

Recall: Our High-Level Design



Pausing/Quitting

- Design option 1: Treat the control commands just like game moves. Return them to the driver (class Game) to handle
- Design option 2: Control commands can be handled (mostly) by the input module itself (e.g., after pausing, ignore all the moves until resuming)
- We'll explore both possibilities a bit

Option 1: Returning all Commands

- Previously, in Game::play, we invoked input.nextMove(), which returned a move_t type element
- Now, we can return a command, which is either a game move, or a control command

```
enum ctrl_t {pause, resume, quit, confirm, cancel, foo, bar};
struct command_t { bool isMove;  move_t mv; ctrl_t ctl; };
```

Wasteful: Only one of the two members mv and ctl will be used

```
struct command_t {
  bool isMove;
  union {move_t mv; ctrl_t ctl; };
};
```

Can refer to either member .mv and .ctl, but overlapping memory locations are used (and one of them is typically meaningless)

Option 1: Returning all Commands

- Handle the various control commands

```
command_t cmd = input.nextCommand();
if(!cmd.isMove) {
  ctrl_t c = cmd.ctl;
  if(c==pause) {
    // handle pausing
  } else if (c==resume) {
    // handle resuming
  } else if (c==quit) {
    // handle quit request
  } else if (c==confirm) {
    // handle quit confirmation
  } else if (c==cancel) {
    // handle cancelling quit
  } else // throw exception
}
```

```
switch(c) {
  case pause:
      // handle pausing
      break;
  case resume:
      // handle resuming
      break;
  case quit:
      // handle quit request
      break;
  case confirm:
      // handle quit confirmation
      break;
  case cancel:
      // handle cancelling quit
      break;
  default:
      // throw exception
}
```

Option 1: Returning all Commands

- Handle the various control commands

```
command_t cmd = input.nextCommand();
if(!cmd.isMove) {
  ctrl_t c = cmd.ctl;
  if(c==pause) {
    // handle pausing
  } else if (c==resume) {
    // handle resuming
  } else if (c==quit) {
    // handle quit request
  } else if (c==confirm) {
    // handle quit confirmation
  } else if (c==cancel) {
    // handle cancelling quit
  } else // throw exception
}
```

If paused, no need to continue the loop until resumed

Then no need for separate handling of `resume` command, as it will be consumed as part of handling `pause`

Will need to invoke a method in the `input` object to handle pause

`input` needn't have returned the `pause` command at all, as it will be asked to handle it right after that (and `Game::play()` gets cluttered)

Option 2: Control Commands not Returned

- Can handle pause + resume (and quit + cancel) within the input module, without involving the driver
  - Driver will use `input.nextMove()` as before, which, if paused, will return only after resuming
- How about quit + confirm?
  - Can use exception handling to quit
  - try/catch in `main()`

```
int main() {
  game G;
  try {
   int s = G.play();
  } catch(QuitException q) {
    cout << "Bye!" << endl;
  }
}
```

class QuitException {};

Option 2: Control Commands not Returned

- Input module should display prompts (e.g. "Press Enter to resume")
- Give it a pointer to the drawer, which offers a method to display information during the game
- boardDrawer uses a new class infoPanel

```
class infoPanel {   // boardDrawer will have an object of this class as a member
  //...
  Rectangle panel;  // simplecpp graphics object
  Text infoText;    // simplecpp graphics object
public:
  //...
  void updateInfo(const string& txt);  // change the text
  void showScore();                    // change the text to show score
  void updateScore(int s, int xrows);  // change the internally stored score
};
```

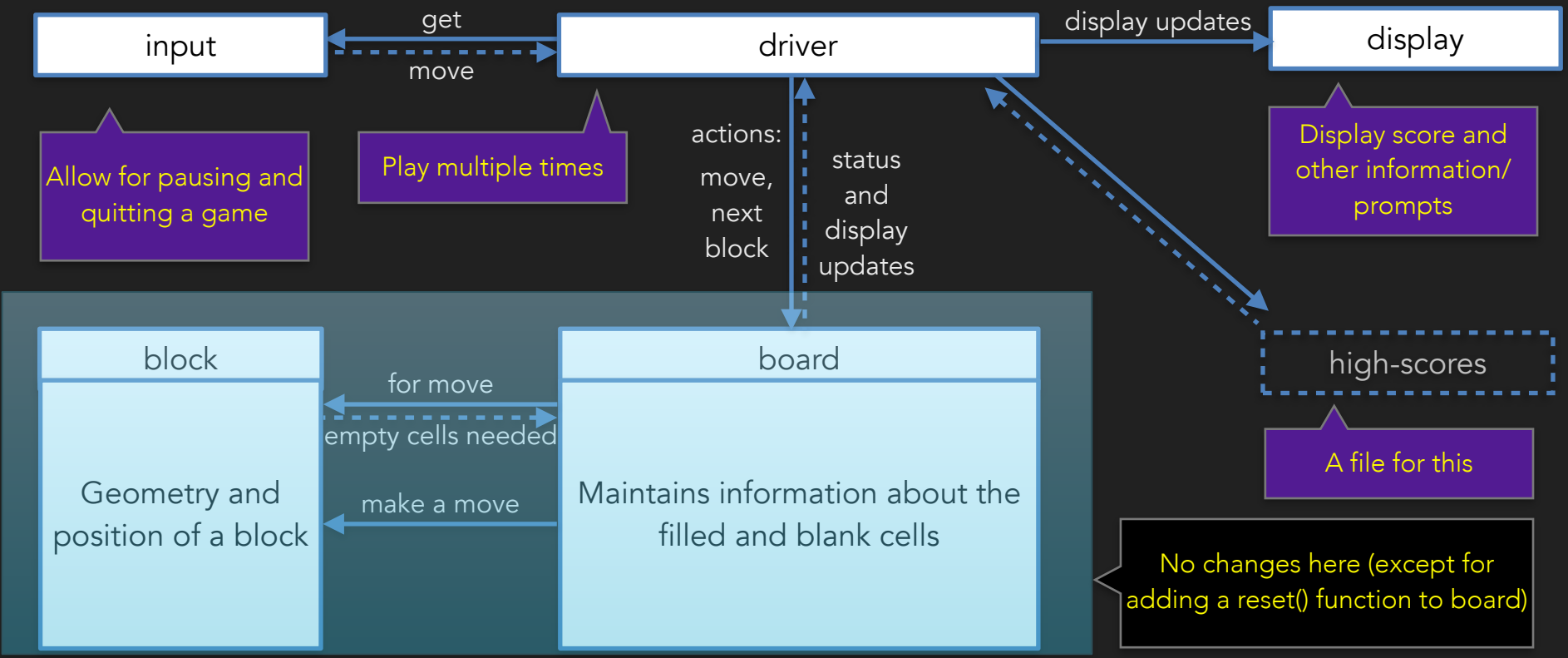
Option 2: Control Commands not Returned

- Quitting and pausing within the input module

```
// Checks for quitting (with confirmation),
// if so throws QuitException
// Returns true if updateInfo called
bool quit(XEvent& ev) {
  if(!isQuit(ev)) return false;
  brdDrw->updateInfo("Quit? Enter/Esc");
  do {
    nextEvent(ev);  // this is a "blocking" call
  } while(!isConfirm(ev) && !isCancel(ev));
  if(isConfirm(ev)) throw QuitException();
  brdDrw->updateInfo("Not quitting");
  return true;
}
```

```
move_t nextMove() {
  XEvent ev;
  while (checkEvent(ev) ) {
    if(pause(ev) || quit(ev)) {
      brdDrw->showScore();
      continue;
    }
    try {
      return moveFromKey(ev);
    } catch(BadKeyException) {}
  }
  return none;
}
```

Recall: Our High-Level Design



Adding a Loop

```
int main() {
  game G;
  try {
    while(1) {
      int s = G.play();
    }
  } catch(QuitException q) {
      cout << "Bye!" << endl;
  }
}
```

```
int game::play() {
  //...
  Tet.reset(); // to play again on the same board
  drawer.refresh();
  input.start();
  while(Tet.newBlock(nextBlock(),falling)) {
    //...
  }
  return score
}
```

```
void board::reset() {
  for(auto& r : blocks)
    std::fill(r.begin(),r.end(),nil);
}
```

```
void start() {
  XEvent ev;
  brdDrw->updateInfo("Enter to start");
  do { nextEvent(ev); } while(!isConfirm(ev));
  brdDrw->showScore();
}
```

A History File

```
int main() {
  history H(SCOREFILENAME);
  game G;
  try {
    while(1) {
      int s = G.play();
      if(H.eligible(s)) {
        record r;
        r.setTime();
        r.score = s;
        cout << NAMEPROMPT;
        std::getline(cin,r.name);
        H.addRecord(r);
      }
    }
  } catch(QuitException q) { /*...*/ }
}
```

```
struct record {
  int score;
  time_t when;
  string name;
  bool operator< (const record& r) const;
  void setTime() {
    typedef std::chrono::system_clock clk;
    when = clk::to_time_t(clk::now());
  }
};
```

```
class history {
  string filename;
  int maxsize;
  std::set<record> records; //sorted
public:
  //constr, destr, eligible(), addRecord()
};
```

A History File

```
history::history(const string& fname, int n=10) : filename(fname), maxsize(n) {
  std::ifstream infile (filename);
  while(infile) {
    record r; infile >> r;
    if(infile) records.insert(r);
  }
  if(records.size() > maxsize)  // let us not shrink the size
    maxsize = records.size();
}
void history::flush() {
  std::ofstream outfile (filename,ios::trunc);
  if(outfile)
    for (auto& r : records)
      outfile << r;
}
history::~history() { flush(); }
```

A History File

```
void history::addRecord(record& r) {
  records.insert(r);
  if(records.size() > maxsize)
    records.erase(--records.end()); // the "lowest" is the max under operator<
}

bool history::eligible(int score) {
  return (records.size() < maxsize || (records.rbegin())->score < score);
}
```

Conclusion

- Programming is ubiquitous: Simple "scripts" to complex cyber-physical systems
- C++ is just one of several programming languages out there
- Belongs to the "C-like" family
  - Which itself is a very large family: C, Java, Perl, Javascript, PHP, C#, Go, Rust, Swift, ...
- Also many other popular languages
  - Python, Ruby, FORTRAN, Visual Basic, SQL, MATLAB, Haskell, ...
- It is likely that you will learn/use more languages over time

Conclusion

- C++ is just one of several programming languages out there
  - It is likely that you will learn/use more languages over time
- C++ itself has several features we have not seen in this course
  - E.g., support for concurrent programming
- Also, several powerful libraries
- Several tools to help with building good software
  - Compiler already can do a lot (e.g., optimisation)
  - Debugging tools, "profiling tools", build automation tools, "Copilots", formal verification tools, ...

Conclusion

- Programming is ubiquitous
- Many powerful programming tools
  - languages, features, libraries, and other tools to create safer, faster software more efficiently (and adapt it quickly when needed)
- CS 101 is just the start of your programming journey

Bon Voyage!