INTRO TO C PROGRAMMING (USING C++)

LECTURE 01-2

JIM FIX, REED COLLEGE CS2-F20

TODAY

Let's take a look at some basic C(++) programming

- Program "anatomy": **`main`** with supporting functions & procedures
- Program "statements":
  - variables & types, assignment, I/O, **`if-else`**, loops, **`return`**

No textbook for this, only my notes & examples; references.

- Reading: Stroustrup's "Tour" Ch1; "PrPrC++ " Chs 5-6,9-10,12
- (Note: Harvard's popular "CS50" has notes & tutorials on C)
- My (limited) grammar for C++, along with your knowledge of Python

ANATOMY OF A C PROGRAM

a preamble of `#include` lines for needed *header files*

a *procedure* or *function* definition

a *procedure* or *function* definition

...

a *procedure* or *function* definition

definition of the `main` function

EXAMPLE: helloImOut.cc

```
#include <iostream>

int main(void) {

  std::cout << "Hello, world!\n";
  std::cout << "I must be going...\n";

  return 0;
}
```

EXAMPLE: cToF.cc

```
#include <iostream>

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  int f = c * 9 / 5 + 32;
  std::cout << "That's " << f << " degrees fahrenheit.\n";

  return 0;
}
```

RUNNING A C PROGRAM

- Python is an *interpreted* language, you run *another* program **`python3`** run it.
  - ➡ The command **`python3`** is the Python *interpreter*. It is a *machine executable*.

```
python3 myProgram.py
```

- C code isn't normally run with an interpreter.

- C programs are *compiled* using another program, like so:

```
g++ -o myProgram myProgram.cc
```

  - The command **`g++`** is a C++ compiler. It is also a machine executable.
  - The file named **`myProgram.cc`** is a C program's source code.
  - The compiler produces a file named **`myProgram`** which is a machine executable, too!

- So, technically speaking, you don't directly run a C program.

RUNNING A C PROGRAM'S EXECUTABLE

- A machine executable file is a sequence of bytes.
  - These bytes make up the codes of machine-readable instructions.
  - They are "written" (by the compiler) in the machine's *language*.

- To run a machine executable named **`myProgram`** type the command line:

  `./myProgram`

  - The **`./`** is the "file path" to the program (the folder where it lives) .
    - ➡ The notation **`.`** here means *"this folder that you're working within."*

- **NOTE:** the files **`myProgram`**, **`python3`**, **`g++`** are all machine executable files.
  - ➡ And they probably all were compiled from C++ source code!

INSPECTING MACHINE CODE

- Just for fun, we can use the Unix editor emacs to inspect code:
  - If I type the command below, I get to see its bytes.

```
emacs myProgram
```

- You can also write machine code in a machine's *assembly language*.
- The C compiler can also write that assembly code for you:

```
g++ -S myProgram.cc
```

- The line above produces a human-readable (well, -ish) file named **`myProgram.s`**.
- This is normally either "x86 code" in either *AT&T style*, or else *Intel style*, assembly.

BACK TO SYNTAX: ANATOMY OF MAIN

- Every C program has a main function.
- It must have the form shown below*:

```
int main(void) {
```

Sequence of ***program statements*** *that the program should perform when run, in order of their execution;*
This is called the ***body*** *of the* **`main`** *function.*

```
    return 0;†
}
```

* We'll later learn how to use parameters **`argc`** and **`argv`** for **`main`** instead of **`void`**.

† We'll at some point learn how to return a non-zero (error) value here.

PROGRAM STATEMENTS

- The main function has a body of statements. Example statements include:
  - A variable declaration e.g. `int c;`
    - syntax is ***type-name variable-name*** `;`
  - A variable assignment e.g. `f = c*9/5+32;`
    - syntax is ***variable-name*** `=` ***expression*** `;`
  - Input of a value into a variable from the `std::cin` input *stream*
    - syntax is `std::cin >>` ***variable-name*** `;`
  - Output of text and values to the `std::cout` output *stream*
    - syntax is `std::cout <<` ***expression*** `;`
  - return of a value. Syntax is `return` ***expression*** `;`
- Like Python, there are also conditionals and loops.

EXAMPLE: guess.cc

```
#include <iostream>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()

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

  int number = (rand() % 100) + 1;
  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";
  return 0;
}
```

EXAMPLE: guess.cc

```
#include <iostream>
#include <ctime>     // For time()
#include <cstdlib>   // For srand() and rand()

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

  int number = (rand() % 100) + 1;
  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";
  return 0;
}
```

variable declarations

VARIABLES & TYPES

- In C++, a variable names a place in memory that stores a value as a sequence of bits/bytes.
- The representation depends on the *type* of its data.
  - E.g. a **`char`** is only one byte, i.e. 8 bits
- The type **`int`** is for integer values. It is four bytes wide, i.e. 32 bits.
  - values ate -2^31 up to +2^31-1
  - **`unsigned int`** has same length, but values are 0 up to 2^32-1
  - **`long`** has twice the length, eight bytes wide
- The type **`double`** is for floating-point values, i.e. "calculator values"
  - it uses eight bytes, +/-2.3E-308 to +/-1.7E+308
  - **`float`** is four bytes long, +/-1.2E-38 to +/-3.4E+38, less precision, use **`double`**

- **NOTE:** variable use must be consistent, can't mix the use. *Strictly enforced.*

EXAMPLE: guess.cc

```
#include <iostream>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()

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

  int number = (rand() % 100) + 1;
  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";
  return 0;
}
```

while loop

EXAMPLE: guess.cc

```
#include <iostream>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()

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

  int number = (rand() % 100) + 1;
  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";
  return 0;
}
```

output statement

EXAMPLE: guess.cc

```
#include <iostream>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()

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

  int number = (rand() % 100) + 1;
  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";
  return 0;
}
```

"includes" list

EXAMPLE: guess.cc

```
#include <iostream>
#include <ctime>     // For time()
#include <cstdlib>   // For srand() and rand()

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

  int number = (rand() % 100) + 1;
  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";
  return 0;
}
```

"includes" list

some uses of items defined in the includes

EXAMPLE: guess.cc

```
#include <iostream>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()

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

  int number = (rand() % 100) + 1;
  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";
  return 0;
}
```

EXAMPLE (CONT'D): THE LOOP BODY FOR guess.cc

```
while (!success) {
  std::cin >> guess;
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else if (guess > number) {
    std::cout << "That's too high. Try again.\n";
  } else {
    success = true;
  }
}
```

EXAMPLE (CONT'D): THE LOOP BODY FOR guess.cc

conditional statement

```
while (!success) {
  std::cin >> guess;
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else if (guess > number) {
    std::cout << "That's too high. Try again.\n";
  } else {
    success = true;
  }
}
```

EXAMPLE (CONT'D): THE LOOP BODY FOR guess.cc

```
while (!success) {
  std::cin >> guess;
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else if (guess > number) {
    std::cout << "That's too high. Try again.\n";
  } else {
    success = true;
  }
}
```

input statement

C VS. PYTHON SO FAR

- A C program is a collection of program components
  - It is not a line-by-line, top-to-bottom script.
  - Instead, it is a series of *declarations*.
    - A declaration defines each component: a function, a new type, etc.
  - Must have a **`main`** function defined amongst its components.
  - **`main`** is the top-level description of what the program does.

- C variables have to be explicitly defined before they get used.
  - There needs to be a declaration of their type.
  - A variable's use has to be uniform in type.

C VS. PYTHON SO FAR (CONT'D)

- A C program is not run by an interpreter; it is compiled instead.
  - We use `g++ -o pgm pgm.cc` to make a program named `pgm`.

- Whitespace (tabs, spacing, ends of lines) don't matter to C.
  - Use braces `{}` and semicolons `;` to structure code.
  - We format carefully only for readability.

- Comments are either
  - embedded `/* like this */` within a line, or they are
  - at the end of a line `// like this.`

GRAMMAR FOR C PROGRAMS: FUNCTIONS AND CODE BLOCKS

program ::= *function-declarations main*

main ::= `int main(void) {` *block* `}`

block ::= *variable-declarations statements*

statement ::=

variable-name = *expression* ;

`std::cin >>` *variable-name* ;

`std::cout <<` *expression* ;

conditional

loop

update ;

`return` *expression* ;

GRAMMAR FOR C PROGRAMS: VARIABLE DECLARATIONS

statement ::=

variable-name = *expression* ;

…

variable-declaration ::=

type-name *variable-name* ;

type-name *variable-name* = *expression* ;

type-name *variable-name* { *expression* };

type-name ::=

`int` | `double` | `bool` | `char` | `std::string` | …

GRAMMAR FOR CONDITIONALS, WHILE, UPDATES

conditional ::=

`if (` *expression* `) {` *block* `}`

`if (` *expression* `) {` *block* `} else {` *block* `}`

`if (` *expression* `) {` *block* `} else if (` *expression* `) {` *block* `}`

`if (` *expression* `) {` *block* `} else if (` *expression* `) {` *block* `} else {` *block* `}`

…

loop ::=

`while (` *expression* `) {` *block* `}`

`do {` *block* `} while (` *expression* `);`

`for (` *statement* `;` *expression* `;` *statement* `) {` *block* `}`

update ::=

variable-name *operation*`=` *expression* `;`

variable-name`++;` | *variable-name*`--;` | `++`*variable-name*`;` | `--`*variable-name*`;`

GRAMMAR FOR EXPRESSIONS

expression ::=

expression binary-operation expression

unary-operation expression

literal-value

variable-name

...

binary-operation ::= `+` | `-` | `*` | `/` | `%` | `&&` | `||` | `<` | `==` | `>` | `<=` | `>=` | `!=`

unary-operation ::= `-` | `!`

literal-value ::= `3` | `3.14159` | `true` | `"hello"` | `'c'` | `...`

variable-name ::= `x` | `y0` | `doThis` | `or_this` | `...`

A FEW THINGS TO TRY

- Install and run the **Atom** editor.
  - Has a GitHub component (an Atom "plug-in").
  - Has a collaboration component.
  - Has a ssh/ftp component.
- Try out the on-line C++ system on **repl.it**
  - ➡ Link is https://repl.it/languages/cpp
- Login to one of the "*dumplings*", CS-managed Linux machines:

```
ssh jimfix@gyoza.reed.edu
```

UPCOMING COURSE WORK

- Thursday/tomorrow: Will publish a **Homework 01** on the web page.
  - ➡ A few simple C++ program puzzles.
  - ➡ Work from my examples, low-pressure assignment.
  - ➡ Can use repl.it, Atom, dumpling, whatever works.
- Next Tuesday: **Lab 02** assignment
  - ➡ Practice with Unix commands, Git, Unix editing and compilation.
  - ➡ Will teach you how to obtain & submit assignments from GitHub.
- Next Wednesday: continue with C++
  - ➡ You can ask questions about Homework 01
  - ➡ You can ask questions about submitting through GitHub
- Next Thursday: *Homework 01 due*

GRAMMAR for expressions

***statement* ::= *var-name* = *expression*; | ...**

***expression* ::=**

***expression binary-op expression* ;**

***unary-op expression*;**

(*expression*)

literal-value

var-name

binary-operation* ::= *arithmetic* | *comparison* | *logical

***arithmetic* ::= + | * | – | / | %**

***comparison* ::= == | < | <= | > | >= | !=**

***logical* ::= && | ||**

GRAMMAR for expressions

***statement* ::= *var-name* = *expression*; | ...**

***expression* ::=**

expression binary-op expression ;

unary-op expression;

(expression)

literal-value

var-name

binary-op* ::= *arithmetic* | *comparison* | *logical

arithmetic* ::= *+* | *** | *–* | */* | *%

comparison* ::= *==* | *<* | *<=* | *>* | *>=* | *!=

logical* ::= *&&* | *||

GRAMMAR for expressions

***statement* ::= *var-name* = *expression*; | ...**

***expression* ::=**

expression binary-op expression ;

unary-op expression;

(expression)

literal-value

var-name

binary-op* ::= *arithmetic* | *comparison* | *logical

***arithmetic* ::= + | * | – | / | %**

***comparison* ::= == | < | <= | > | >= | !=**

***logical* ::= && | ||**

GRAMMAR for expressions

***statement* ::= *var-name* = *expression*; | ...**

***expression* ::=**

***expression binary-op expression* ;**

***unary-op expression*;**

(*expression*)

literal-value

var-name

binary-op* ::= *arithmetic* | *comparison* | *logical

***arithmetic* ::= + | * | – | / | %**

***comparison* ::= == | < | <= | > | >= | !=**

***logical* ::= && | ||**

GRAMMAR for expressions

statement **::=** ***var-name*** **=** ***expression*** **; | ...**

expression **::=**

expression binary-op expression **;**

unary-op expression **;**

(***expression*** **)**

literal-value

var-name

literal-value **::= `42` / `3.14` / `true` / `"hello"` / `'A'` / ...**

unary-op **::= - / !**

GRAMMAR for a C program

program **::=** ***includes defs main***

main **::=** **`int main(void) {`** ***block*** **`}`**

block **::=** ***statements***

statement **::=**

var-name = expression;

var-dec

`std::cout <<` ***outs*** **`;`**

`std::cin >>` ***var-name*** **`;`**

`return` ***expression*** **`;`**

conditional

loop

update

Example: guess.cc

```
#include <iostream>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()

int main() {

  srand(time(0));

  int number = (rand() % 100) + 1;

  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";

  return 0;
}
```

Example: guess.cc loop

```
while (!success) {
  std::cin >> guess;
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else if (guess > number) {
    std::cout << "That's too high. Try again.\n";
  } else {
    success = true;
  }
}
```

Example: guess.cc

```
#include <iostream>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()

int main() {

  srand(time(0));

  int number = (rand() % 100) + 1;

  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";

  return 0;
}
```

Example: guess.cc loop

```
while (!success) {
  std::cin >> guess;
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else if (guess > number) {
    std::cout << "That's too high. Try again.\n";
  } else {
    success = true;
  }
}
```

If and If-Else Statements

- Just like in Python, you can use conditional statements to perform execution of code blocks, driven by certain checks.

- There is an "*if*" statement

```
if(condition-to-test)  {
    statements-to-execute-if-true
}
```

- There is an "*if-else*" statement

```
if(condition-to-test)  {
    statements-to-execute-if-true
} else {
    statements-to-execute-if-false
}
```

- **Note:** no semicolon after the brace for these "compound" statements.

Cascading "if" statements

- They can end with just an "*if*"

```
if(condition-to-test) {
    statements-to-execute-if-true
} else if (some-other-test) {
    statements-to-execute-for-this-test
} else if...

} else if (...) {
    statements
}
```

- They can end with an "*else*"

```
if(condition-to-test) {
    statements-to-execute-if-true
} else if...

} else {
    statements
}
```

Loops!

- Just like in Python, you can use loops to perform *iteration, i.e. r*epeated execution of a block of code until some condition no longer holds.

- There is an "***while***" statement

```
while(condition-to-test) {
    statements-to-execute-when-true
}
```

- There is also a "***do-while***" statement!!!!

```
do {
    statements-to-execute-once-and-continue-by-test-below
} while(condition-to-test);
```

countUp.cc using while

```
#include <iostream>

int main() {

  int top;
  std::cout << "Enter the ending count: ";
  std::cin >> top;

  int count = 0;
  while (count <= top) {
    std::cout << count << "\n";
    count = count++;
  }
  std::cout << "Woo!\n";

  return 0;
}
```

countDown.cc using while

```
#include <iostream>

int main() {

  int top;
  std::cout << "Enter the starting count: ";
  std::cin >> top;

  int count = top;
  while (count > 0) {
    std::cout << count << "\n";
    count = count--;
  }
  std::cout << "Woo!\n";

  return 0;
}
```

More C-like using for

```
#include <iostream>

int main() {

  int top;
  std::cout << "Enter the ending count: ";
  std::cin >> top;

  for (int count = 0; count <= top; count++) {
    std::cout << count << "\n";
  }
  std::cout << "Woo!\n";

  return 0;
}
```

More C-like using for

```
#include <iostream>

int main() {

  int top;
  std::cout << "Enter the starting count: ";
  std::cin >> top;

  for (int count = top; count > 0; count--) {
    std::cout << count << "\n";
  }
  std::cout << "Woo!\n";

  return 0;
}
```

"for" loop

- Anytime you have a loop like this

```
initial-statement
while(condition-to-test) {
    statements-to-execute-when-true
    update-statement
}
```

- You can write it like below

```
for(initial-statement ; condition-to-test; update-statement) {
    statements-to-execute-when-true
}
```

GRAMMAR for a C program

program **::=** ***fundefs main***

main **::=** **`int main(void) {`** ***block*** **`}`**

block **::=** ***vardefs statements***

statement ::=

- ***variable*** **`=`** ***expression*** **`;`**
- ***update*** **`;`**
- **`std::cout <<`** ***outs*** **`;`**
- **`std::cin >>`** ***in*** **`;`**
- **`return`** ***expression*** **`;`**
- ***conditional***
- ***loop***

GRAMMAR for C statements

conditional ::=

`if (`*expression*`) {`*block*`}`
`if (`*expression*`) {`*block*`} else {`*block*`}`
`if (`*expression*`) {`*block*`} else if (`*expression*`) {`*block*`}`
`if (`*expression*`) {`*block*`} else if (`*expression*`) {`*block*`} else {`*block*`}`
...

loop ::=

`while (`*expression*`) {`*block*`}`
`do {`*block*`} while (`*expression*`);`
`for (`*statement*`;`*expression*`;`*statement*`) {`*block*`}`

***update* ::=**

variable operation = expression ;
***variable* `++;`**
***variable* `--;`**

Example: guess.cc

```
#include <iostream>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()

int main() {

  srand(time(0));

  int number = (rand() % 100) + 1;

  std::cout << "I've chosen a number from 1 to 100. ";
  std::cout << "Try to guess what it is.\n";

  int guess;
  bool success = false;

  while (!success) {
    ... // keep getting guesses and reporting their success
  }

  std::cout << "Well done! ";
  std::cout << number << " was the number I chose.\n";

  return 0;
}
```

Example: guess.cc loop

```
while (!success) {
  std::cin >> guess;
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else if (guess > number) {
    std::cout << "That's too high. Try again.\n";
  } else {
    success = true;
  }
}
```

Code with style!

You should start getting in the habit of

...Using comments either **`/*as blocks*/`** or **`//at line ends`**

...Indenting nicely to make code readable.

...Using good function/variable names

- in **`camelCaseForm`** or **`snake_case_form`**

...Breaking your code up into meaningful procedures and functions.

See my **`guessGame.c`** for an example, including top comments.

Coming up…

- I will post a Homework 01 for you to accept and start.
- TAs can help you install XCode or Ubuntu/WSL pn your machine.
- Next week:
  - ➡ procedures and functions
  - ➡ strings, conversions, more on I/O (?)
  - ➡ structs and arrays
  - ➡ some general hackery (with & and *)