Review

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: January 4, 2025

Basic program structure, local/global variables, value passing semantics, strings, program dev. process

Hello world!

Hello world!

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

int main( ){
  cout << "Hello, World!" << endl;
  return 0;
}
```

- 4 types of control structures:

Hello world!

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

int main( ){
  cout << "Hello, World!" << endl;
  return 0;
}
```

- 4 types of control structures:
- sequences (see above)

Hello world!

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

int main( ){
  cout << "Hello, World!" << endl;
  return 0;
}
```

- 4 types of control structures:
- sequences (see above)
- conditionals

Hello world!

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

int main( ){
  cout << "Hello, World!" << endl;
  return 0;
}
```

- 4 types of control structures:
- sequences (see above)
- conditionals
- loops

Hello world!

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

int main( ){
  cout << "Hello, World!" << endl;
  return 0;
}
```

- 4 types of control structures:
- sequences (see above)
- conditionals
- loops
- function invocations

Functional abstraction

Functional abstraction

Functional abstraction

Functional abstraction

```
// Declaration of the triple function
int triple(int x);

int main( ){
  int answer;
  answer = triple(5);
  cout << answer << endl;
  cout << triple(2) << endl;
  return 0;
}

// Definition of the triple function
int triple(int x) {
  return 3 * x;
}
```

Functional abstraction

```
// Declaration of the triple function
int triple(int x);

int main( ){
  int answer;
  answer = triple(5);
  cout << answer << endl;
  cout << triple(2) << endl;
  return 0;
}

// Definition of the triple function
int triple(int x) {
  return 3 * x;
}
```

- Must declare functions before referencing them

Functional abstraction

```
// Declaration of the triple function
int triple(int x);

int main( ){
  int answer;
  answer = triple(5);
  cout << answer << endl;
  cout << triple(2) << endl;
  return 0;
}

// Definition of the triple function
int triple(int x) {
  return 3 * x;
}
```

- Must declare functions before referencing them
- use function prototype /header

Functional abstraction

```
// Declaration of the triple function
int triple(int x);

int main( ){
  int answer;
  answer = triple(5);
  cout << answer << endl;
  cout << triple(2) << endl;
  return 0;
}

// Definition of the triple function
int triple(int x) {
  return 3 * x;
}
```

- Must declare functions before referencing them
- use function prototype /header
- OR declare before 1st use

Functional abstraction

```
// Declaration of the triple function
int triple(int x);

int main( ){
  int answer;
  answer = triple(5);
  cout << answer << endl;
  cout << triple(2) << endl;
  return 0;
}

// Definition of the triple function
int triple(int x) {
  return 3 * x;
}
```

- Must declare functions before referencing them
- use function prototype /header
- OR declare before 1st use
- Scope of a function = file scope

Functional abstraction

```
// Declaration of the triple function
int triple(int x);

int main( ){
  int answer;
  answer = triple(5);
  cout << answer << endl;
  cout << triple(2) << endl;
  return 0;
}

// Definition of the triple function
int triple(int x) {
  return 3 * x;
}
```

- Must declare functions before referencing them
- use function prototype /header
- OR declare before 1st use
- Scope of a function = file scope
- Can a function call itself?!

Local and global variables and constants

Local and global variables and constants

Local and global variables and constants

Local and global variables and constants

```
// Declaration of a global variable
int g;

// Declaration of a global constant
const int THREE = 3;

int main( ){
  const int LOC = 29;
  int loc = LOC;
  g = 42;
  cout << g << endl;
  tripleGlobal();
  cout << g << endl;
  return 0;
}
```

Local and global variables and constants

```
// Declaration of a global variable
int g;

// Declaration of a global constant
const int THREE = 3;

int main( ){
  const int LOC = 29;
  int loc = LOC;
  g = 42;
  cout << g << endl;
  tripleGlobal();
  cout << g << endl;
  return 0;
}
```

Local and global variables and constants

```
// Declaration of a global variable
int g;

// Declaration of a global constant
const int THREE = 3;

int main( ){
  const int LOC = 29;
  int loc = LOC;
  g = 42;
  cout << g << endl;
  tripleGlobal();
  cout << g << endl;
  return 0;
}
```

```
void tripleGlobal( ){
  // The local var loc is not acc.
  // The global var g is accessible
  g = THREE * g;
}
```

- Use “extern” to access global variables declared in other files

Conditionals (if-then-else branching)

Conditionals (if-then-else branching)

```
int max(int a, int b){
  if (a >= b)
    return a;
  else
    return b;
}



int main( ){
  cout << max(-1, 2) << endl;
  cout << max(1, -2) << endl;
  return 0;
}
```

Conditionals (ternary operator cond ? b1 : b2)

- Compare the following:

Conditionals (ternary operator cond ? b1 : b2)

- Compare the following:

Conditionals (ternary operator cond ? b1 : b2)

- Compare the following:

```
int max(int a, int b){
  if (a >= b)
    return a;
  else
    return b;
}

int max(int a, int b) {
  return (a >= b) ? a : b;
}
```

Conditionals (nesting)

- Can be nested:

Conditionals (nesting)

- Can be nested:

Conditionals (nesting)

- Can be nested:

```
int inRange(int num, int low, int high) {
  if(num>=low)
    if(num<=high)
      return 1;
  return 0;
}
```

- Note: could have used a compound conditional statement instead

Conditionals (else-if and switch cases)

- Can have multiple branches:

Conditionals (else-if and switch cases)

- Can have multiple branches:

Conditionals (else-if and switch cases)

- Can have multiple branches:

```
int sign(int a){
  if (a > 0)
    return 1;
  else if (a < 0)
    return -1;
  else
    return 0;
}
```

Conditionals (else-if and switch cases)

- Switch cases?

Conditionals (else-if and switch cases)

- Switch cases?

Conditionals (else-if and switch cases)

- Switch cases?

```
switch (month){
 case 1: case 2: case 3: case 4:
   cout << "Winter";
   break;
 case 5: case 6: case 7: case 8:
   cout << "Spring";
   break;
 case 9: case 10: case 11: case 12:
   cout << "Fall";
   break;
 default:
   cout << "What are we smoking today?";
 }
```

Repetition structures (loops)

- Want to compute:

Repetition structures (loops)

- Want to compute:
- f(n) = 1 + 2 + 3 + … + n

Repetition structures (loops)

- Want to compute:
- f(n) = 1 + 2 + 3 + … + n

Repetition structures (loops)

- Want to compute:
- f(n) = 1 + 2 + 3 + … + n

```
unsigned int triangular(unsigned int n){
  unsigned int result = 0;
  for (unsigned int i = 1; i <= n; i++){
    result += i;
  }
  return result;
}
```

- Order of execution?

Repetition structures (loops)

- Want to compute:
- f(n) = 1 + 2 + 3 + … + n

```
unsigned int triangular(unsigned int n){
  unsigned int result = 0;
  for (unsigned int i = 1; i <= n; i++){
    result += i;
  }
  return result;
}
```

- Order of execution?
- Can have an empty body!

Repetition structures (loops)

Repetition structures (loops)

```
const unsigned int BASE = 10;

unsigned int sumOfDigits(unsigned int m){
  unsigned int sum = 0;
  while (m != 0) {
    unsigned int digit;
    digit = m % BASE;
    sum = sum + digit;
    m = m / BASE;
  }
  return sum;
}
```

- Trace it!

Repetition structures (loops)

```
const unsigned int BASE = 10;

unsigned int sumOfDigits(unsigned int m){
  unsigned int sum = 0;
  while (m != 0) {
    unsigned int digit;
    digit = m % BASE;
    sum = sum + digit;
    m = m / BASE;
  }
  return sum;
}
```

- Trace it!
- More explicit than for loops

Repetition structures (loops)

```
const unsigned int BASE = 10;

unsigned int sumOfDigits(unsigned int m){
  unsigned int sum = 0;
  while (m != 0) {
    unsigned int digit;
    digit = m % BASE;
    sum = sum + digit;
    m = m / BASE;
  }
  return sum;
}
```

- Trace it!
- More explicit than for loops
- Do-while: like while, but executes at least once

Repetition structures (loops)

```
const unsigned int BASE = 10;

unsigned int sumOfDigits(unsigned int m){
  unsigned int sum = 0;
  while (m != 0) {
    unsigned int digit;
    digit = m % BASE;
    sum = sum + digit;
    m = m / BASE;
  }
  return sum;
}
```

- Trace it!
- More explicit than for loops
- Do-while: like while, but executes at least once

Value passing semantics

- Call by value (arguments evaluated)

Value passing semantics

- Call by value (arguments evaluated)

Value passing semantics

- Call by value (arguments evaluated)

```
void doubleV(int a){
  a = a*2;
}

int main( ){
  int a = 2;
  doubleV(a+a);
  cout << a << endl;

  return 0;
}
```

Value passing semantics

- Call by reference (can only send vars)

Value passing semantics

- Call by reference (can only send vars)

Value passing semantics

- Call by reference (can only send vars)

```
void doubleR(int &a){
  a = a*2;
}

int main() {
  int a = 4;
  doubleR(a);
  cout << a << endl;

  return 0;
}
```

Value passing semantics

- Call by address (arguments evaluated)

Value passing semantics

- Call by address (arguments evaluated)
  - We'll see more of this later

Value passing semantics

- Call by address (arguments evaluated)
  - We'll see more of this later
  - Have to explicitly get dereference

Value passing semantics

- Call by address (arguments evaluated)
  - We'll see more of this later
  - Have to explicitly get dereference
    - i.e. get value from the address

Value passing semantics

- Call by address (arguments evaluated)
  - We'll see more of this later
  - Have to explicitly get dereference
    - i.e. get value from the address

Value passing semantics

- Call by address (arguments evaluated)
  - We'll see more of this later
  - Have to explicitly get dereference
    - i.e. get value from the address

```
void doubleP(int *a){
  *a = (*a)*2;
}

int main( ){
  int a = 4;
  doubleP(&a);
  cout << a << endl;

  return 0;
}
```

Side effects

- Effects of a function other than the generation of a value to be returned

Side effects

- Effects of a function other than the generation of a value to be returned
  - those that persist

Side effects

- Effects of a function other than the generation of a value to be returned
  - those that persist
- e.g., printing stuff using cout, changing a global variable, changing a local variable via call by reference/pointer, etc.

Strings

- C++ libraries provide string facilities

Strings

- C++ libraries provide string facilities

Strings

- C++ libraries provide string facilities

```
#include <string>

int main( ){
  string h = "hello";
  string w = "world";
  string msg = h + ' ' + w;
  cout << msg << endl;
  return 0;
}
string s = "hello world";
for (int i = 0; i < s.length(); i++)
  cout << s[i] << endl;
```

Strings

- Characters are integer values

Strings

- Characters are integer values

Strings

- Characters are integer values

```
char charToUpper(char c){
  if ('a' <= c && c <= 'z')
    return c - 'a' + 'A';
  else
    return c;
}
```

Strings

- Passing by reference: faster than pass-by-value for large strings

Strings

- Passing by reference: faster than pass-by-value for large strings
- Not safe: modifying the passed string also modifies the original one

Strings

- Passing by reference: faster than pass-by-value for large strings
- Not safe: modifying the passed string also modifies the original one
- Solution: pass by constant reference

Strings

- Passing by reference: faster than pass-by-value for large strings
- Not safe: modifying the passed string also modifies the original one
- Solution: pass by constant reference

Strings

- Passing by reference: faster than pass-by-value for large strings
- Not safe: modifying the passed string also modifies the original one
- Solution: pass by constant reference

```
string capitalize(const string &s);
```

- Occasionally, you may want to return a value by constant reference (meh!)

Strings

- Passing by reference: faster than pass-by-value for large strings
- Not safe: modifying the passed string also modifies the original one
- Solution: pass by constant reference

```
string capitalize(const string &s);
```

- Occasionally, you may want to return a value by constant reference (meh!)

Strings

- Passing by reference: faster than pass-by-value for large strings
- Not safe: modifying the passed string also modifies the original one
- Solution: pass by constant reference

```
string capitalize(const string &s);
```

- Occasionally, you may want to return a value by constant reference (meh!)

```
const string &chooseFirst(const string &s1, const string &s2) {
   if (s1 < s2)
     return s1;
   else
     return s2;
 }
```

Code as Communication

- Passing by constant reference doesn’t add any power to the language

Code as Communication

- Passing by constant reference doesn't add any power to the language
  - We can do *less* things with a const reference

Code as Communication

- Passing by constant reference doesn't add any power to the language
  - We can do *less* things with a const reference
- This is **good**

Code as Communication

- Passing by constant reference doesn't add any power to the language
  - We can do *less* things with a const reference
- This is **good**
- Code communicates an intention

Code as Communication

- Passing by constant reference doesn't add any power to the language
  - We can do *less* things with a const reference
- This is **good**
- Code communicates an intention
  - "This function shouldn't change this string"

Code as Communication

- Passing by constant reference doesn't add any power to the language
  - We can do *less* things with a const reference
- This is **good**
- Code communicates an intention
  - "This function shouldn't change this string"
- Compiler *checks* this intention

Code as Communication

- Passing by constant reference doesn't add any power to the language
  - We can do *less* things with a const reference
- This is **good**
- Code communicates an intention
  - "This function shouldn't change this string"
- Compiler *checks* this intention
  - Gives you an error if you violate it

Strings

Strings

```
string &chooseFirst(string &s1, string &s2)
{
  if (s1 < s2)
    return s1;
  else
    return s2;
}
```

- chooseFirst( ) returns reference to lexicographically smaller string

Strings

```
string &chooseFirst(string &s1, string &s2)
{
  if (s1 < s2)
    return s1;
  else
    return s2;
}
```

- chooseFirst( ) returns reference to lexicographically smaller string
- main( ) prints PQR! since s1=PQR!

Strings

Strings

```
int main( )
{
  string s1 ="ABC!";
  string s2 = "XYZ!";

  chooseFirst(s1, s2) = "PQR!";

  cout << s1;

  return 0;
}
```

Modular vs. Application programs (115 vs. 110)

- Top-down design

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools
  - eventually solve original problem

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools
  - eventually solve original problem
- Reuse

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools
  - eventually solve original problem
- Reuse
  - reduces the overhead of solving a problem over and over again,

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools
  - eventually solve original problem
- Reuse
  - reduces the overhead of solving a problem over and over again,
  - saves us from redoing testing and documentation for similar code

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools
  - eventually solve original problem
- Reuse
  - reduces the overhead of solving a problem over and over again,
  - saves us from redoing testing and documentation for similar code
  - Easier to understand code

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools
  - eventually solve original problem
- Reuse
  - reduces the overhead of solving a problem over and over again,
  - saves us from redoing testing and documentation for similar code
  - Easier to understand code
  - Code structured into modules

Modular vs. Application programs (115 vs. 110)

- Top-down design
  - repeatedly decomposing a complicated problem into smaller, easier subproblems
  - each can be implemented independently
  - e.g., decomposing a function into many smaller ones
- Alternative is bottom-up approach
  - building reusable tools
  - then using those tools to build even powerful tools
  - eventually solve original problem
- Reuse
  - reduces the overhead of solving a problem over and over again,
  - saves us from redoing testing and documentation for similar code
  - Easier to understand code
  - Code structured into modules
    - separates interface from implementation

Standard input and output

- Can redirect standard input and output from and to files resp.

Standard input and output

- Can redirect standard input and output from and to files resp.
- `myProg < inFile > outFile`

Standard input and output

- Can redirect standard input and output from and to files resp.
- `myProg < inFile > outFile`
- Can pipe the standard output of a program to the standard input of another

Standard input and output

- Can redirect standard input and output from and to files resp.
- `myProg < inFile > outFile`
- Can pipe the standard output of a program to the standard input of another
- `myProg1 | myProg2`

Standard input and output

- Can redirect standard input and output from and to files resp.
- `myProg < inFile > outFile`
- Can pipe the standard output of a program to the standard input of another
- `myProg1 | myProg2`
- See notes for how

Standard input and output

- Can redirect standard input and output from and to files resp.
- `myProg < inFile > outFile`
- Can pipe the standard output of a program to the standard input of another
- `myProg1 | myProg2`
- See notes for how
- `getline(cin, <string>)` and `cin.get(<char>)` can be used to read input from a file

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together
  - `g++ -c main.cpp`

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together
  - `g++ -c main.cpp`
  - `g++ -c my_util.cpp`

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together
  - `g++ -c main.cpp`
  - `g++ -c my_util.cpp`
  - `g++ -o prog.out main.o my_util.o`

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together
  - `g++ -c main.cpp`
  - `g++ -c my_util.cpp`
  - `g++ -o prog.out main.o my_util.o`
  - Collect all function prototypes together in a header file and include it in main.cpp

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together
  - `g++ -c main.cpp`
  - `g++ -c my_util.cpp`
  - `g++ -o prog.out main.o my_util.o`
  - Collect all function prototypes together in a header file and include it in main.cpp

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together
  - `g++ -c main.cpp`
  - `g++ -c my_util.cpp`
  - `g++ -o prog.out main.o my_util.o`
  - Collect all function prototypes together in a header file and include it in main.cpp

```
#include "my_util.h"
#pragma once preprocessor
```

- Assertions (debugging aid)

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together
  - `g++ -c main.cpp`
  - `g++ -c my_util.cpp`
  - `g++ -o prog.out main.o my_util.o`
  - Collect all function prototypes together in a header file and include it in main.cpp

```
#include "my_util.h"
#pragma once preprocessor
```

- Assertions (debugging aid)

Misc

- Separate (unrelated) functions in different files; compile separately using -c command, and link together
  - `g++ -c main.cpp`
  - `g++ -c my_util.cpp`
  - `g++ -o prog.out main.o my_util.o`
  - Collect all function prototypes together in a header file and include it in main.cpp

```
#include "my_util.h"
#pragma once preprocessor
```

- Assertions (debugging aid)

```
#include <cassert>
...
assert (n>0); //prog. Terminates if not
```