Scientific programming with C++

INPUT AND OUTPUT, VARIABLES, FUNCTIONS, PARAMETERS

Learning objectives

- After this lecture and related assignments, you will…
  - know how to use variables in common cases
    - convert them to other variables
    - and the implications of using different data types
  - have basic knowledge of what functions are
    - the difference between passing by value and passing by reference
    - and the difference between parameter and argument
  - be introduced to declarations, definitions and initializations
  - know how to read input from the user
  - know how to print output to the user

Input and output basics

- Input: data flows from device (keyboard) to memory
- Output: data flows from memory to device (computer screen)
- Main headers
  - iostream
    - For reading keyboard input and outputting to command window
  - fstream
    - For reading files and writing files
- This topic will focus on iostream

Output: cout

- The basic building block of printing to screen: cout
  - Part of iostream header, std namespace

```
std::cout << "Whatever you want to print";
```

- Continues from where the previous stream ended

```
std::cout << "This is one string.";
std::cout << "This follows immediately after.";
```

```
This is one string.This follows immediately after.
```

- Insert line breaks with std::endl or "\n"
- Chain strings or other variables into the output stream with <<

```
int number = 2; std::string text = "This is a string.";
std::cout << text << std::endl << "Here's string number " << number << ".";
```

```
This is a string.
Here's string number 2.
```

Input: cin

- The basic building block of reading keyboard input: cin
  - Like cout, part of iostream and std
- Prompts the user to write something in the console
- Uses the >> operator to save the input stream to a variable
  - Terminated with the enter button

```
// create variable "myInput"
std::string myInput;
// write keyboard input to "myInput"
std::cin >> myInput;
```

- Written variable doesn't necessarily have to be a string

```
int myNumber;
std::cin >> myNumber;
```

Variables

- Store data values
- Lots of different data types
  - int: integers (whole numbers)
  - double: floating-point numbers
  - bool: Boolean values (true or false)
  - char: single character
  - string: several characters together

```
int a = 2;
double b = 0.45; double c = 1. / 7.;
bool understood = false;
char initial = 'W'; char time = 't';
std::string asdf = "I like trains.";
```

Data types and memory usage

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-9223372036854775808 to 9223372036854775807
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 18446744073709551615
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

Conversion between variables

- Known as type-casting
  - Implicit conversion

```
int a = 3;
float b;
b = a;
```

  - Explicit conversion

```
int a = 3;
float b;
b = (float)a; // c-like casting
b = float(a); // functional conversion
```

- Various other functions

```
int a = 11;
// the following requires the "string" header
// int to string
std::string numberString = std::to_string(a);
// string back to int
int b = std::stoi(numberString);
```

Global vs local variables

- Normally variables (like most entities) are local to the block they are in and its nested blocks
- Variables can be declared at the top of the file before the main() function to make them accessible anywhere in the program
  - Global variables
- Relevant keywords:
  - extern
    - Tells the compiler that the variable exists (usually in another file)
  - static
    - The value of the variable persist through different calls of a function or instances of a class

References

- Aliases for existing variables
- Created by putting an ampersand (&) between variable type and name when creating variables

```
int a = 10; // creates an integer
int& b = a; // creates a reference to a
int &c = a; // also creates a reference to a
a = 20;
std::cout << "a: " << a << std::endl; // variable a prints value of a
std::cout << "b: " << b << std::endl; // reference of a prints value of a
std::cout << "c: " << c << std::endl; // reference of a prints value of a
std::cout << "&a: " << &a << std::endl; // note: pointer, not reference
```

```
a: 20
b: 20
c: 20
&a: 0000004E1515F854
```

Pointers

- Variables that store memory addresses of other variables
- Created by putting an asterisk (*) between variable type and name when creating variables and using the ampersand operator before a declared name
- Dereference operator (also an asterisk) returns the underlying value

```
int a = 10;
int* b = &a;
a = 20;
std::cout << "a: " << a << std::endl; // variable a prints value of a
std::cout << "b: " << b << std::endl; // variable b (pointer) prints address of a
std::cout << "&a: " << &a << std::endl; // directly print the pointer
std::cout << "*b: " << *b << std::endl; // dereference pointer to get the value
```

```
a: 20
b: 000000E8208FF4E4
&a: 000000E8208FF4E4
*b: 20
```

References vs pointers

- Once initialized to an object, references cannot be changed
  - Pointers can be changed to point to another object

```
int a = 1;
int c = 2;
int& b = a;
b = c; // not a reassignment of the
reference; rather, the value of c is
assigned to the object referred to by b
std::cout << "a: " << a << std::endl;
std::cout << "b: " << b << std::endl;
std::cout << "c: " << c << std::endl;
```

```
a: 2
b: 2
c: 2
```

```
int a = 1;
int c = 2;
int* b = &a; // b is a pointer to a
std::cout << "b: " << b << std::endl;
b = &c; // change b to point to c instead of a
std::cout << "b: " << b << std::endl;
```

```
b: 000000687E7FF904
b: 000000687E7FF924
```

References vs pointers

- References must be initialized (given an initial value) when they are created, pointers can be initialized later

```
int& a; // error
int* b; // valid
```

- You cannot have null references (references not pointing anywhere), but you can have null pointers

```
int& a = NULL; // error
int* b = NULL; // valid
```

- Pointers point to the memory address of a variable, references to the value directly

References & pointers: Why bother?

- References (and pointers) allow you to pass arguments to functions…
  - without copying the entire object that is passed as argument
  - while allowing to modify the object given as argument inside the function
  - while utilizing polymorphism (more in a later topic)
- More about references in functions in later slides this lecture
- Prefer references over pointers
  - Breaking your program is easier with pointers
  - Use pointers only when you must

Functions

- Perform an operation or several operations
- Must be declared and defined before using
- Using a function is “calling” or “invoking” it
- Both user-written and built-in
- Parameters
- Arguments
- Return type

Declaration vs definition vs initialization

- Declaration introduces a new entity to the compiler
  - Variable, function, type, class, whatever
  - "This identifier exists"
  - For functions, specifies signature
    - Name, return type and parameters
- Definition allocates memory for the entity
  - "This identifier works like this"
  - For functions, defines its body
    - What operations the function does
- Initialization assigns an initial value to entity
- Variable definitions are often also declarations
- Variable declaration-definitions can also be initializations

```
double inverse(double input);
```

```
double inverse(double input) {
    return (1.0 / input);
}
```

```
int a; // declaration and definition: identifier "a" exists and is an integer
int b = 1; // declaration, definition and initialization
```

Functions: parameter vs argument

```
double num1 = -6;
double num2 = 2.5;
double product;
product = productFunction(num1, num2);
```

- Argument:
  - Value that is passed to the function when the function is called
- Parameter:
  - Belongs to the function signature
  - Defined when the function is declared
  - Becomes a local variable of the function
    - Is assigned the value of the argument

```
bool productFunction(double a, float b)
{
    return a*b;
}
```

- A function doesn't need to have inputs or outputs

Functions: return type

- Many functions have a return type that is specified in function declaration

```
bool isPositive(double num)
{
    return num > 0;
}
```

- Functions without a return type are given the void "type"

```
void sayHello() {
    std::cout << "Hello" << std::endl;
}
```

  - The above example also has no parameters (and no arguments when it's called)

Functions: pass by value and pass by reference

- Pass by value
  - The argument is copied into the scope of the function
    - Can be expensive if the argument is a large object
  - Cannot modify values outside the function
  - Safer choice
- Pass by reference: put an ampersand before parameter name
  - The function gets a reference to the object instead copying the object itself
  - Modifies the value of the argument even outside the function
  - Faster choice when arguments are large objects
- The same function can have some parameters passed by value and some passed by reference

Functions: pass by value vs pass by reference

```
int plusThreeValue(int num) {
    num += 3; // increment 3
    return num;
}
```

```
int plusThreeReference(int& num) {
    num += 3; // increment 3
    return num;
}
```

```
int myNumber = 1;
int outputByValue = plusThreeValue(myNumber); // returns 4
// myNumber is still 1, because it was copied into the function
int outputByReference = plusThreeReference(myNumber); // returns 4
// myNumber is now 4, because 3 was added to myNumber instead of its copy
```

Functions: returning values by reference

- Put an ampersand between function type and name
- Allows functions to be used on the left side of an assignment operator
- Never return a local variable outside its scope!

```
// an array of doubles
double globalArray[] = {1.4, 6.0, 4.2};

// function to return by reference
double& updateArray(unsigned int i) {
    return globalArray[i];
}

int main()
{
    // set the 3rd element to 1.0
    updateArray(2) = 1.0;
}
```

```
double& returnByReferenceLocal() {
    double someDouble = 23.5;
    return someDouble;
}
```

Function templates

- Functions that can operate with generic instead of specific types
- Works with several data types or classes, but with just one definition
- Return types and parameters can be templatized
- Eliminates the need for many similar functions

Function templates: parameter

```
template <typename whateverType>
void printArray(whateverType inArray, unsigned int numElements) {
    for (unsigned int i = 0; i < numElements; ++i) {
        std::cout << inArray[i] << " ";
    }
    std::cout << std::endl;
}

int main()
{
    int intArray[4] = {5, 70, 12, 56};
    float floatArray[3] = {1.5, 50.3, -12.4};
    printArray(intArray,4);
    printArray(floatArray,3);

}
```

```
5 70 12 56
1.5 50.3 -12.4
```

Function templates: return type

```
template <typename whateverType>
whateverType additiveInverse(whateverType a) {
    return -a;
}

int main()
{
    int a = 2;
    float b = 2.5;
    double c = -2546.05;
    std::cout << additiveInverse(a) << std::endl;
    std::cout << additiveInverse(b) << std::endl;
    std::cout << additiveInverse(c) << std::endl;
}
```

```
-2
-2.5
2546.05
```

Command-line arguments

- The main function can receive arguments from the command line

```
int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; ++i)
        std::cout << argv[i] << std::endl;

}
```

```
>Variables-functions.exe 1 2.3 hello
Variables-functions.exe
1
2.3
hello
```

Summary

- Don’t be intimidated by all the new words
- Knowing when to pass by value or reference (or pointer) and what data types to use for your variables affects performance
  - Limit memory usage
  - Process data faster
- Pointers should be avoided
  - But understanding them is required for many further concepts in C++