

CIS 330

C/C++ and Unix

Lecture 7

Introduction to C++

The C++ logo is displayed in white text on a solid blue rectangular background. The logo consists of a large 'C' followed by two '+' signs.

Key differences to C

New features

- templates

- overloading

- inheritance

- useful data structures (Vectors, Strings)

Compiling C++ Code

Name your files with **cc**, **cxx**, or **cpp** extension (or anything that makes sense, just be **consistent**)

- Google coding standard suggest **cc**

Same goes for the header files

- **.h** works, but you can also use **.hpp** (again, just be consistent)
- **.h** is still the most popular and follows the Google coding standard

We will use **.cc** and **.h** for this course

Compile your code using **g++**

- Still part of the GNU compiler collection
- **'g++'** is equivalent to **'gcc -xc++ -lstdc++ -shared-libgcc'**
 - **xc++** is the compiler option, and
 - the others are linker options (C++ links to different libraries)

Can I mix C and C++

You can call C code in C++ **or** call C++ code in C using this method

- Simple answer – yes

```
extern "C" void func(int);  
or  
extern "C" {  
    Int func2();  
    double func3();  
}
```
- Syntax for C and C++ are very similar – e.g., you can use pointers in C++ as well
- But, it's better not to mix (if you have the choice).
- You can find more details here:
 - <https://isocpp.org/wiki/faq/mixing-c-and-cpp>

I/O

`cin` – standard input (`stdin`)

`cout` – standard output (`stdout`)

`cerr` – standard error (`stderr`)

`clog` – standard logging, typically used for non-critical events

Output

```
std::cout << "Hello World\n";
```

- Standard (std) character output device (cout)
- Insertion operator (<<) – insert/send the following to the cout

```
std::cout << "i is " << i << std::endl;
```

Output

```
#include <iostream>
```

```
#include <iomanip>
```

```
double j = 1.23456;
```

```
std::cout << std::fixed;
```

```
std::cout << setprecision(2);
```

```
std::cout << "j is " << j << std::endl;
```

Or use printf (or fprintf)

Input

```
int a;  
  
std::cout << "Please enter an integer" << endl;  
  
std::cin >> a;
```

or

```
int a;  
  
int b;  
  
std::cout << "Please enter two integers" << endl;  
  
std::cin >> a >> b;
```

- This method consider white space as termination
- Like in the case with C, entering a non-integer value for int can result in unpredictable behavior

Namespace

Group names into a narrower scope – allows organizing elements of program into different *logical* scope

- Useful for avoiding collisions for generic names

```
using namespace std;
```

- Allows the components of std – standard C++ library – to be visible

```
using namespace std;
```

```
cout << "i is " << i << endl;
```

Variable Initialization

```
int a = 5; /* Regular */  
int b(6); /* constructor init */  
int c{9}; /* uniform init */
```

- Uniform init requires new C++ standard
- This makes it easier to differentiate because {} and () – functional form
- This will make more sense when we start talking about classes and objects (and constructors)

Control Flow

- If-then
- While
- Do-While
- For
- Switch
- `break` and `continue`
- `goto`

Range-based For Loop

Iterates over all the elements in a *range*, where *declaration* is some variable that can take the value of a single element

```
for(declaration : range) {  
    statements;  
}
```

For example,

```
string str = "Hello World";  
for(char c : str) {  
    std::cout << c << " ";  
}  
  
std::cout << endl;
```

Data Types

- char, int, float, double, etc. and its usual specifies (e.g., unsigned, long, etc.)
- boolean (`bool`) - standard in C++ (i.e., built-in data type), but not in C (but an implementation was provided in later versions of C)

“New” Data Types

Type deduction using `auto`

```
int a = 7;
```

```
auto b = a;
```

Type deduction using `decltype`

```
decltype(a) c;
```

`auto` vs. `decltype`

- `decltype` gives you the declared type for variable `c`
- `auto` uses template deduction (matters for reference/pointers, discussed later)

Pointers

- Pretty much the same between C and C++
- C++ provides an additional way to pass an address into a function - pass by reference using & (instead of a pointer)

```
void f(int& r) {  
    cout << "r = " << r << endl;    /* print the value inside r  
*/  
    cout << "&r = " << &r << endl; /* address of r */  
    r = 5;                          /* value of r has changed  
                                   permanently */  
    cout << "r = " << r << endl;  
}  
  
int main() {  
    int x = 47;  
    f(x);  
}
```

Efficiency

Value vs. Reference - which is more efficient?

```
string concatenate(string a, string b)
{
    /* a and b are copied - i.e., space required for the copies */
    return (a + b);
}
```

```
string concatenate_2(string &a, string &b)
{
    /* a and b are references - they can unintentionally be
    changed

    In this function */
    return (a + b);
}
```


Safety

Can you keep the efficiency of pass-by-reference and prevent the content from being changed?

```
string concatenate_2(const string &a, const string &b)
{
    return (a + b);
}
```

More on Variables

Global variables

- Defined outside of all functions and available to all parts of the program (even in code in other files)
- Local variables
 - Local to a function
 - Also known as ***automatic*** variables (automatically created & destroyed).

Register variables

- Type of local variable ***hint*** that tells the compiler that it should be stored in a register (not guaranteed)
- Must be declared within a block (i.e., cannot be global)
- Does not have an address

More on Variables

static keyword

- Does not disappear even when the scope for the variable is over (e.g., outside of a function it was defined in)
- The variable “lives” for the entire life of the program
- Can only be initialized once and retains its value between function calls
- Why not global variable?
 - Scope is still the same - it cannot be “seen” outside of the function it was defined in - minimizes errors

```
void func() {  
    static int i = 0;  
    cout << "i = " << ++i << endl;  
}  
  
int main() {  
    for(int x = 0; x < 10; x++)  
        func();  
}
```

More on Variables

`extern` keyword

- Tells the compiler that this variable (or function) exists even if it has not seen it yet (e.g., may be defined in another file it has not gotten around to compiling)
- Also used to let C++ know that some C functions exist (in another .c file)
- It can also be used to tell the compiler that the variable is defined further down the same file

```
extern int i;
extern void func();
int main() {
    i = 0;
    func();
}

int i;
void func() {
    i++;
    cout << i;
}
```

More on Variables

`const` keyword

- The variable is **constant** (i.e., it cannot be changed)
- Why not use `#define`?
 - `#define` is simple text replacement - it has no scope or type checking
 - Better than `#define` (unless you want to use it for more than just defining a variable - e.g., a function-like behavior)

More on Variables

`volatile` keyword

- Tells the compiler you do not know when this variable will change (kind of like an opposite to `const`)
- Prevents the compiler from performing any optimizations
- For example, in a multi-threaded program this variable may be used to communicate between threads
 - Even if it appears that it has not changed since it was defined, it likely has
 - Always read the value whenever it is needed (even if it was read just a line before for something else)

Operators

Pretty much the same as C

- Mathematical operators (+, -, *, /, %)
- Relational (they produce a bool data type) - `true` or `false`
 - `<`, `>`, `<=`, `>=`, `==`, `!=`
 - Used with all built-in types
- Logical operators (`&&`, `||`)
 - Used by both `int` and `float/double`
 - Note that for `float/double`, comparison with `0` is absolute, but the smallest difference in the numbers will be considered “not equal.”
- Bitwise operators (`&`, `|`, `^`, `~`)
- Shift operators (`<<`, `>>`)
 - Be careful when using with signed vs. unsigned

Operators

Unary operators (-, +)

Ternary operators

- Remember that `A ? B : C` is different from `if-then` in that it produces a value (i.e., an operator)

```
a = --b ? b : (b = -99);
```

Casting operators

- Compiler usually change data types automatically (e.g., `int` to `float` if you assign an `int` to a `float` variable)
- You can also do it manually (e.g., `(unsigned long int) b` where `b` is a regular `int`)

Operators

Casting should be done **carefully and rarely (or avoid if possible)**, as it can lead to dangerous behavior and prone to bugs (by bypassing the compiler's **automatic type checking**)

C++ specific casting (explicit cast)

- Allows you to identify casting more easily
- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast` (this will be discussed later)

static_cast

Used for conversions that are well-defined (i.e., safe conversions that the compiler would allow)

castless conversion

```
int i = 0x7fff; long l;
```

```
l = i; or l = static_cast<long>(i);
```

narrowing

```
i = static_cast<int>(l);
```

conversion from void*

```
void* vp = &i;
```

```
fp = static_cast<float*>(vp); /* still dangerous! */
```

const_cast

Convert from `const` to **not** `const` or from `volatile` to **not** `volatile`

```
const int i = 0;

int* j = (int*)&i; // Deprecated form

j = const_cast<int*>(&i); // Preferred
```

If you take the address of a `const` object, you produce a ***pointer to a `const`*** and this cannot be assigned to a non-`const` pointer (without a cast)

However, if you try to change the content of `i` (e.g., in another function), it will produce an **undefined behavior**, because C++ does not allow this behavior (changing what is determined to be `const`)

More on `const` later in the class

reinterpret_cast

Cast a completely different meaning - VERY DANGEROUS

Typically used to do some bit twiddling, and will NEED to be cast back to the original type before doing anything else

```
struct X { int a[sz]; };  
  
X x;  
  
int* xp = reinterpret_cast<int*>(&x);  
  
for(int* i = xp; i < xp + sz; i++)  
    *i = 0;  
  
/* must be cast back to struct X before being used as X  
*/  
  
/* reinterpret_cast<X*>(xp) */
```

sizeof

This is actually an operator

- Used to get the size of data-types
- Also used to get size of user-defined data types in C++ (discussed later)

asm

- Mechanism that allows you to write assembly code “embedded” into your C/C++ code
- Easier alternative to disassembling your code, changing it, and then reassembling it
- We will (probably) have a class on how to write simple assembly code

Functions

C++ allows optional/default parameters

```
void opt_param(int i, int j = 10);
```

- You can pass in just the first parameter `i` or both `i` and `j`.
- If `j` is not passed in, it assumes the value of 10

Overloading

Two different functions can have the same name – if they have different number of parameters, or different data type parameters

```
void overload(int i, int j);
```

```
void overload(double i, double j);
```

```
void overload(int i, int j, int k);
```

However, avoid having functions with the same name do different things (it gets confusing)

But, useful if you want to have functions that may have different inputs (e.g., integer or double, optional n^{th} input, etc.)

Templates

What if you have a function whose parameters can be of numerous different types?

Have an overloaded function for each type

Or use a template

```
template <class mytype> mytype sum(mytype a, mytype b);
```

```
int x = sum<int>(10, 20);
```

```
double y = sum<double>(10.1, 20.1);
```

Compiler automatically *instantiates* a version of the `sum` function with `mytype` being replaced by `int`, `double`, `float`, etc.

Composite Type Creation

typedef - works as in C

```
typedef unsigned long ulong;
```

structs

you can use it as in C

structs plays a more important role in C++ (classes evolved from structs, as we will see later)

enum - works as in C

Automatically assigns values to names

You can also manually assign values to names

```
enum ShapeType {  
    circle = 10, square = 20, rectangle = 50  
};
```

union - works as in C

Memory saving mechanism

Strings

C++ provides a String class

```
#include <string>
```

Works similar to C (but more convenient, will be discussed later)

String Stream (stringstream)

Similar to Strings (again, will be discussed later)

```
#include <sstream>

string stream_str = "5081";

int input_int;

stringstream(stream_str) >> input_int;

std::cout << "Input_int is " << input_int << endl;
```

Vectors

- Strings for numbers – good for dynamically sized arrays
- Vectors class is a template – it can be efficiently applied to different data types

Vector Example

```
1.  vector<string> vec;
2.  ifstream fin("lecture09b.cc");
3.  string line;
4.  while(getline(fin, line)) {
5.      vec.push_back(line);
6.  }
7.  for(int i = 0; i < vec.size(); i++) {
8.      cout << i << ": " << vec[i] << endl;
9.  }
```

Live Coding