C++ Basics

Rapidly, for senior devs

Overview

- Type System
- Storage types (heap, free, static)
- Declarations and definitions
  - Operator and function overloading
  - Argument Dependent Lookup (ADL)
- Classes
  - Constructor/Destructor, RAII
- Argument passing
- Function and class templates
- Not covered (yet): Exception Safety

Not covered… (yet)

- Exception safety
- Const correctness
- Initialization
  - default-, zero-, constant-, static -

Storage Types

- Automatic
  - Preferred for ‘small’ objects
  - Scope-based
  - Value typed!
- Free
  - std::make_unique<T>
- Static
  - From program start to program end
  - ! mind static initialization fiasco !

Type System

- Statically typed!
- NOT portable (but implementation defined)
  - Hence C++11: std::uint32_t, ...
- Extensible (through classes)
- Promotion rules
  - Unsigned always wins
  - Floating points always win
  - Compiler warnings unless explicit static_cast<...>

Declarations and Definitions

Compilation times are high…. :(

=> developer can tell what is strictly needed for code to be _used_

- Header file (.hpp, .h, .hxx): “Interface” and storage requirements
  - forward declarations
  - class interface + members
  - Function prototypes
  - !! Must be on include path of client code => conflicts possible!
- Implementation file (.cpp, .cxx)
  - repeats prototype + adds implementation

Declarations and Definitions

- Operator and function overloading
  -
    ```
    ostream operator<<(ostream &o, MyBigInt i){
      return o << i.text;
    }
    ...
    MyBigInt myInt("9999999999999999999");

    auto s = myInt + 1000 * MyBigInt("5e100");
    cout << "there's a big int: " << s << endl;
    ```
  -

Declarations and Definitions: ADL

Argument Dependent Lookup (ADL, Koenig Lookup)

Function lookup scheme for ‘unqualified’ names:

- Namespace of arguments
- Then: ‘normal’ lookup

More info: cppreference

Usage: generic algorithms

```
namespace x {
    struct A {};
    void foo(A){}
}
void foo(int){}

A a;
foo(a); // results in x::foo(a)
```

Declarations and Definitions: ADL

Usage

- Generic algorithms: use unqualified functions to allow extension
    - Remember “Open for Extension?”
    - Example: use begin(c) iso c.begin() to allow array arguments, too

```
template<class C>
void printCont(const C& c){
  copy(begin(c), end(c),
    ostream_iterator...(cout));
}

int arr[10];
printCont(arr);

vector<int> vec(10);
printCont(vec);
```

Classes

- Groups ‘stateful’ functions together
- Special members:
    - construct/destruct: A a(1, 2, 3);
    - copy-constructor, move-constructor: A a2(a); A a3(A(1, 2, 3));
    - copy-assignment, move-assignment: a = a3; a2 = A(1, 2, 5);
- Overloadable operators
    - +, -, +=, <<, >>, |, |=, …..
    - new
- Access levels
    - public, protected, private

Classes: RAII

‘Resource Allocation Is Initialization’

Usage: for ‘scoped lifetime’, avoid code sandwiches

“C++ does not need garbage collection because it does not produce garbage”

```
class File {
  FILE* p;
public:
  File(string path): p(fopen(path))
  {}
  ~File(){ fclose(p); }
};

{
    File f("abcd.txt");
}
```

Argument Passing

- Default: pass by value
- Big/Expensive data: pass by const reference
- Output arguments: pass by value
- Pointers… what’s that?

Function and class templates

- Compile-time genericity
- Phases
  - Template declaration `template<class T> void foo(T t);`
  - Template definition `template<class T> void foo(T t) {}`
  - Template instantiation

    ```
    foo(1);
    foo("abc");
    ```
- Template arguments
  - Can be types, values

    ```
    template<int i> void fac()
    ```

Function templates

- Parameter deduction!
  - Usage: convenient wrappers for extensive template class description:

    ```
    std::tuple<int, const char*> t1 = {1, "abc"};
    auto t2 = std::make_tuple(1, "abc");
    ```
  -

Function and class templates

- Specialization
  - Special cases e.g. vector<bool>
  - 'Metaprogramming' (cpp.sh/25fqk)

    ```
    template<int i> void fac();
    template<> void fac<0>() { return 1; }
    template<int i> void fac() { return i * fac<i-1>(); }
    ```
- Partial Specialization
  - To 'bind' template parameters

    ```
    template<typename T> using shared_vector = std::vector<T, SharedMemoryAlloc>;
    template<typename T> using dict = std::unordered_map<std::string, T>;
    ```
  - :(only for classes

Surprise! Variable templates

Since C++14

Untested…

```
template<int i> int fac;
template<> int fac<0> = 0;
template<int i> int fac = i * fac<i-1>;
```

Further Reading

- https://herbsutter.com/gotw/
  - Novice to guru questions + explanation
  - Reference guy! Bleeding edge c++ standards contributor
- http://en.cppreference.com/w/cpp/language
  - … well… the reference
  - Great support crew

Thanks to/useful online sites

- http://cpp.sh/
- http://markup.su/highlighter/
-