MANCHESTER 1824
The University of Manchester

COMP26020 Programming Languages and Paradigms

Lecture 42: Modern C++

Pavlos Petoumenos
pavlos.petoumenos@manchester.ac.uk

Modern C++: C++11 to C++20

Massive change to the language

- Modules and Concepts (Super important in the long term)
- Threading support (COMP35112)
- defaulted/deleted constructors, List initialisation
- final/override
- any, variant, optional
- Fold expressions
- Attributes
- …and many more…

Modern C++: C++11 to C++20

A subset of changes

- Likely to encounter
- Immediately useful
- Do not fit in another lecture

NULL is bad

`#define NULL 0`

Preprocessor directive

For the compiler there is no such thing as NULL

`if (ptr != NULL)` // comparison between pointer and int

NULL can be redefined!!!

`#undef NULL`
`#define NULL 1` is a valid statement and it will silently break your C code

nullptr

Silently used it already

Language-level keyword

A pointer literal

Implicitly transformed into any pointer type

Works with strong typing, cannot be redefined

auto

Also used it earlier

Lets the compiler deduce types

Super useful with templates, iterators, etc

```
auto it = v.begin();
```

instead of

```
std::vector<int>::iterator it = v.begin();
```

Generic lambdas

```
[](auto x, auto y) { return x > y ? x : y; };
```

auto

Can do more than that

Return type deduction

```
auto fn() { return 0; } // return type is int
auto gn() { return 1.0; } // return type is float
```

Function arguments (abbreviated function template)

```
auto max(auto x, auto y) {….};
```

Types in value templates

```
template <auto n>
auto fn() {return n + n;}
```

C/C++ cannot return multiple values

```
...

int f1(int& other_val) {
     int val1, val2;
     ...
     other_val = val2;
     return val1;
}

int ret1, ret2;
ret1 = f1(ret2);
```

C++ Kinda Containers

std::tuple<Type1, Type2, Type3, ..>

Tuple of as many elements as types in the template

std::pair<Type1, Type2>

Special case of tuple. Easier to use

C/C++ cannot return multiple values

```
...

int f1(int& other_val) {
     int val1, val2;
     ...
     other_val = val2;
     return val1;
}

int ret1, ret2;
ret1 = f1(ret2);
```

Return a pair!


```
#include <pair>
...

std::pair<int, int> f1(int& other_val) {
     int val1, val2;
     ...
     other_val = val2;
     return val1;
}

int ret1, ret2;
ret1 = f1(ret2);
```

Return a pair!


```
#include <pair>
...

std::pair<int, int> f1( ) {
     int val1, val2;
     ...
     other_val = val2;
     return val1;
}

int ret1, ret2;
ret1 = f1(ret2);
```

Return a pair!


```
#include <pair>
...

std::pair<int, int> f1( ) {
     int val1, val2;
     ...
     return {val1, val2}; //or std::make_pair(val1, val2);
}


int ret1, ret2;
ret1 = f1(ret2);
```

Structured binding


```
#include <pair>
...

std::pair<int, int> f1( ) {
     int val1, val2;
     ...
     return {val1, val2}; //or std::make_pair(val1, val2);
}



auto [ret1, ret2] = f1();
```

Recap

nullptr

auto

std::pair std::tuple

→ return multiple values

Structured binding

→ decompose composite return values

Up Next

Compile-time evaluation