CS100 Lecture 13

"C" in C++

Contents

"Better C"

C++ was developed based on C.

From *The Design and Evolution of C++*:

> C++ is a general-purpose programming language that
>
> - **is a better C**,
> - supports data abstraction,
> - supports object-oriented programming.

C++ brought up new ideas and improvements of C, some of which also in turn influenced the development of C.

"Better C"

- `bool`, `true` and `false` are built-in. No need to `#include <stdbool.h>`. `true` and `false` are of type `bool`, not `int`.
  - This is also true since C23.
- The return type of logical operators `&&`, `||`, `!` and comparison operators `<`, `<=`, `>`, `>=`, `==`, `!=` is `bool`, not `int`.
- The type of string literals `"hello"` is `const char [N+1]`, not `char [N+1]`.
  - Recall that string literals are stored in **read-only memory**. Any attempt to modify them results in undefined behavior.
- The type of character literals `'a'` is `char`, not `int`.

"Better C"

- `const` variables initialized with literals are compile-time constants. They can be used as the length of arrays.

  ```c
  const int maxn = 1000;
  int a[maxn]; // valid C++, but VLA in C
  ```

- `int fun()` declares a function accpeting no arguments. It is not accepting unknown arguments.
  - This is also true since C23.

Type System

Stronger type checking

Some type conversions (casts) can be very dangerous:

```
const int x = 42, *pci = &x;
int *pi = pci; // Warning in C, Error in C++
++*pi;         // undefined behavior
char *pc = pi; // Warning in C, Error in C++
void *pv = pi; char *pc2 = pv; // Even no warning in C! Error in C++.
int y = pc;    // Warning in C, Error in C++
```

- For `T` $\neq$ `U`, `T *` and `U *` are different types. Treating a `T *` as `U *` is undefined behavior in most cases, but the C compiler gives only a warning!
- `void *` is a hole in the type system. You can cast anything to and from it **without even a warning**.

C++ does not allow the dangerous type conversions to happen **implicitly**.

Explicit Casts

C++ provides four **named cast operators**:

- `static_cast<Type>(expr)`
- `const_cast<Type>(expr)`
- `reinterpret_cast<Type>(expr)`
- `dynamic_cast<Type>(expr)` $\Rightarrow$ will be covered in later lectures.

In contrast, the C style explicit cast `(Type)expr` looks way too innocent.

An ugly behavior should have an ugly looking.

`const_cast`

Cast away low-level constness **(DANGEROUS)**:

```
int ival = 42;
const int &cref = ival;
int &ref = cref; // Error: casting away low-level constness
int &ref2 = const_cast<int &>(cref); // OK
int *ptr = const_cast<int *>(&cref); // OK
```

However, modifying a `const` object through a non-`const` access path (possibly formed by `const_cast`) results in **undefined behavior**!

```
const int cival = 42;
int &ref = const_cast<int &>(cival); // compiles, but dangerous
++ref; // undefined behavior (may crash)
```

`reinterpret_cast`

Often used to perform conversion between different pointer types **(DANGEROUS)**:

```
int ival = 42;
char *pc = reinterpret_cast<char *>(&ival);
```

We must never forget that the actual object addressed by `pc` is an `int`, not a character! Any use of `pc` that assumes it's an ordinary character pointer **is likely to fail** at run time, e.g.:

```
std::string str(pc); // undefined behavior
```

Wherever possible, do not use it!

`static_cast`

Other types of conversions (which often look "harmless"):

```cpp
double average = static_cast<double>(sum) / n;
int pos = static_cast<int>(std::sqrt(n));
```

Some typical usage: ⇒ We will talk about them in later lectures.

```cpp
static_cast<std::string &&>(str) // converts to a xvalue
static_cast<Derived *>(base_ptr) // downcast without runtime checking
```

Minimize casting

[Best practice] Minimise casting. (*Effective* C++ Item 27)

Type systems work as a **guard** against possible errors: Type mismatch often indicates a logical error.

[Best practice] When casting is necessary, **prefer C++-style casts to old C-style casts.**

- With old C-style casts, you can't even tell whether it is dangerous or not!

Type deduction

Functions

Default arguments

Function overloading

In C++, a group of functions can have the same name, as long as they can be differentiated when called.

```cpp
int max(int a, int b) {
  return a < b ? b : a;
}
double max(double a, double b) {
  return a < b ? b : a;
}
const char *max(const char *a, const char *b) {
  return std::strcmp(a, b) < 0 ? b : a;
}
```

Overloaded functions

Overloaded functions should be distinguished in the way they are called.

```
int fun(int);
double fun(int);  // Error: functions that differ only in
                  // their return type cannot be overloaded.
```

```
void move_cursor(Coord to);
void move_cursor(int r, int c); // OK, differ in the number of arguments
```

Overloaded functions

Overloaded functions should be distinguished in the way they are called.

- The following are declaring **the same function**. They are not overloading.

```
void fun(int *);
void fun(int [10]);
```

- The following are the same for an array argument:

```
void fun(int *a);
void fun(int (&a)[10]);
int ival = 42; fun(&ival); // OK, calls fun(int *)
int arr[10];   fun(arr);   // Error: ambiguous call
```

Why?

Overloaded functions

Overloaded functions should be distinguished in the way they are called.

- The following are the same for an array argument:

```
void fun(int *a);
void fun(int (&a)[10]);
int arr[10];   fun(arr);   // Error: ambiguous call
```

  - For `fun(int (&)[10])`, this is **an exact match**.
  - For `fun(int *)`, this involves an array-to-pointer implicit conversion. We will see that this is **also considered an exact match**.

Basic overload resolution

Suppose we have the following overloaded functions.

```
void fun(int);
void fun(double);
void fun(int *);
void fun(const int *);
```

Which will be the best match for a call `fun(a)`?

Basic overload resolution

Suppose we have the following overloaded functions.

```
void fun(int);
void fun(double);
void fun(int *);
void fun(const int *);
```

Obvious: The arguments and the parameters match perfectly.

```
fun(42);   // fun(int)
fun(3.14); // fun(double)
int arr[10];
fun(arr);  // fun(int *)
```

Not so obvious:

```
int ival = 42;
// fun(int *) or fun(const int *)?
fun(&ival);
fun('a');   // fun(int) or fun(double)?
fun(3.14f); // fun(int) or fun(double)?
fun(NULL);  // fun(int) or fun(int *)?
```

Basic overload resolution

```
void fun(int);
void fun(double);
void fun(int *);
void fun(const int *);
```

- `fun(&ival)` matches `fun(int *)`
- `fun('a')` matches `fun(int)`
- `fun(3.14f)` matches `fun(double)`
- `fun(NULL)` ? We will see this later.

Basic overload resolution

1. An exact match, including the following cases:
   - identical types
   - **match through decay of array or function type**
   - match through top-level `const` conversion
2. **Match through adding low-level `const`**
3. Match through integral or floating-point promotion
4. Match through numeric conversion
5. Match through a class-type conversion (in later lectures).

No need to remember all the details. But pay attention to some cases that are very common.

The null pointer

`NULL` is a **macro** defined in standard library header files.

- In C, it may be defined as `(void *)0`, `0`, `(long)0` or other forms.

In C++, `NULL` cannot be `(void *)0` since the implicit conversion from `void *` to other pointer types is **not allowed**.

- It is most likely to be an integer literal with value zero.
- With the following overload declarations, `fun(NULL)` may call `fun(int)` on some platforms, and may be **ambiguous** on other platforms!

```
void fun(int);
void fun(int *);
```

Better null pointer: `nullptr`

In short, `NULL` is a "fake" pointer.

Since C++11, a better null pointer is introduced: `nullptr` (also available in C23)

- `nullptr` has a unique type `std::nullptr_t` (defined in `<cstddef>`), which is neither `void *` nor an integer.
- `fun(nullptr)` will definitely match `fun(int *)`.

  ```
  void fun(int);
  void fun(int *);
  ```

Avoid abuse of function overloading

Only overload operations that actually do similar things. A bad example:

```
Screen &moveHome();
Screen &moveAbs(int, int);
Screen &moveRel(int, int, std::string direction);
```

If we overload this set of functions under the name `move`, some information is lost.

```
Screen &move();
Screen &move(int, int);
Screen &move(int, int, std::string direction);
```

Which one is easier to understand?

```
myScreen.moveHome(); // We think this one!
myScreen.move();
```