CS100 Lecture 15

Function overloading, type conversion

Function overloading

Overloaded functions

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

```
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?

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

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

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

```
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)` ???

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 (later)

`NULL`

`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 since C23)

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

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

Type conversions

Named casts

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

- `static_cast<Type>(expr)`
- `const_cast<Type>(expr)`
- `reinterpret_cast<Type>(expr)`
- `dynamic_cast<Type>(expr)`

`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); // OK, 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); // most likely undefined behavior
```

Wherever possible, do not use it!

`static_cast`

Other types of conversions:

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

Some typical usage (may be talked about in later lectures):

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

Minimize casting

(See *Effective C++* Item 27)

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

- Type systems are very powerful: See *C++ Template Metaprogramming* Chapter 3.1 for an example.

When casting is necessary, **prefer C++-style casts to old C-style casts**.

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