CS100 Lecture 13

"C" in C++

Contents

"C" in C++

- Type System
  - Stronger Type Checking
  - Explicit Casts
  - Type Deduction
- Functions
  - Default Arguments
  - Function Overloading
- Range-Based `for` Loops Revisited

"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 arithmetic conversions are problematic: They are not value-preserving.

```
int x = some_int_value();
long long y = x; // OK. Value-preserving
long long z = some_long_long_value();
int w = z;       // Is this OK?
```

- Conversion from `int` to `long long` is value-preserving, without doubt.
- Conversion from `long long` to `int` may lose precision ("narrowing").

However, no warning or error is generated for such conversions in C.

Stronger type checking

Some arithmetic conversions are problematic: They are not value-preserving.

```cpp
long long z = some_long_long_value();
int w = z; // "narrowing" conversion
```

Stroustrup had decided to ban all implicit narrowing conversions in C++. However,

> The experiment failed miserably. Every C program I looked at contained large numbers of assignments of `int`s to `char` variables. Naturally, since these were working programs, most of these assignments were perfectly safe. That is, either the value was small enough not to become truncated, or the truncation was expected or at least harmless in that particular context.

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, C uses `(Type)expr` for explicit casts.

`const_cast`

Cast away low-level `const`ness **(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;
const int &cref = ival;
int &ref = const_cast<int &>(cref); // compiles, but dangerous
++ref; // undefined behavior
```

`reinterpret_cast`

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

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

We must never forget that the actual object pointed 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.:

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

Wherever possible, do not use it!

`static_cast`

Often used for other types of conversions (which often look "harmless"):

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

We will talk about its more typical usage in later lectures.

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

C++ is very good at **type computations**:

```cpp
std::vector v(10, 42);
```

- It should be `std::vector<int> v(10, 42);`, but the compiler can deduce that `int` from `42`.

```cpp
int x = 42; double d = 3.14; std::string s = "hello";
std::cout << x << d << s;
```

- The compiler can detect the types of `x`, `d` and `s` and select the correct ways to handle them.

`auto`

When declaring a variable with an initializer, we can use the keyword `auto` to let the compiler deduce the type from the initializer.

```
auto x = 42;    // `int`, because 42 is an `int`.
auto y = 3.14;  // `double`, because 3.14 is a `double`.
auto z = x + y; // `double`, because the type of `x + y` is `double`.
auto m;         // Error: cannot deduce the type. An initializer is needed.
```

`auto` can also be used to produce compound types:

```
auto &r = x;          // `int &`, because `x` is an `int`.
const auto &rc = r; // `const int &`.
auto *p = &rc;        // `const int *`, because `&rc` is `const int *`.
```

C23 also has `auto` type deduction.

`auto`

What about this?

```
auto str = "hello";
```

`auto`

What about this?

```cpp
auto str = "hello"; // `const char *`
```

- Recall that the type of `"hello"` is **`const char [6]`**, not `std::string`. This is for compatibility with C.
- When using `auto`, the array-to-pointer conversion ("decay") is performed automatically.

`auto`

Deduction of return type is also allowed (since C++14):

```cpp
auto sum(int x, int y) {
  return x + y;
}
```

- The return type is deduced to `int`.

`auto`

What are the benefits of using `auto`?

Some types in C++ are very long:

```
std::vector<std::string>::const_iterator it = vs.begin();
```

Use `auto` to simplify it:

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

`auto`

What are the benefits of using `auto`?

Some types in C++ are not known to anyone but the compiler:

```
auto lam = [](int x, int y) { return x + y; } // A lambda expression.
```

Every lambda expression has its own type, whose name is only known by the compiler.

`decltype`

`decltype(expr)` will deduce the type of the expression `expr` **without evaluating it**.

```
auto fun(int a, int b) { // The return type is deduced to be `int`.
  std::cout << "fun() is called.\n"
  return a + b;
}
int x = 10, y = 15;
decltype(fun(x, y)) z; // Same as `int z;`.
                       // Unlike `auto`, no initializer is required here.
                       // The type is deduced from the return type of `fun`.
```

- `decltype(fun(x, y))` only deduces the return type of `fun` without actually calling it. Therefore, **no output is produced**.

Functions

Default arguments

Some functions have parameters that are given particular values in most calls. In such cases, we can declare those common values as **default arguments**.

```
std::string get_screen(std::size_t height = 24, std::size_t width = 80,
                       char background = ' ');
```

- By default, the screen is 24×80 filled with `' '`.

  ```
  auto default_screen = get_screen();
  ```

- To override the default arguments:

  ```
  auto large_screen   = get_screen(66);            // 66x80, filled with ' '
  auto larger_screen  = get_screen(66, 256);       // 66x256, filled with ' '
  auto special_screen = get_screen(66, 256, '#'); // 66x256, filled with '#'
  ```

Default arguments

Arguments in the call are resolved by position.

```
auto scr = get_screen('#'); // Passing the ASCII value of '#' to `height`.
                            // `width` and `background` are set to
                            // default values (`80` and `' '`).
```

- Some other languages have named parameters:

  ```
  print(a, b, sep=", ", end="") # Python
  ```

  There is no such syntax in C++.

Default arguments are only allowed for the last (right-most) several parameters:

```
std::string get_screen(std::size_t height = 24, std::size_t width,
                       char background); // Error.
```

Function overloading

In C++, a group of functions can have the same name, as long as they can be differentiated when called (i.e., have different parameters).

```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;
}
```

- Types of parameters are different.

Overloaded functions

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

- Number of parameters is different.

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

- Return type has *no* effect on overloading.

```
int fun(int);
double fun(int);  // Error: differ only in return type.
```

- The following are declaring **the same function**, not overloaded.

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

Overloaded functions

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

- Which function to call is determined by the match between the arguments and parameters.

```
void fun(int a);
void fun(double a);
fun(42); // call fun(int)
fun(3.14); // call fun(double)
```

- The following are the same for an array argument:

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

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 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:
    - no conversion required
    - **match through array decay**
    - 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 (including null pointer 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 value

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

- In C, it may be defined as `0`, `(void *)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 others (e.g., where `NULL = (long)0`)!

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

Better null pointer value: `nullptr`

Since C++11, a better null pointer value 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 type, and can be converted to any pointer type (but not other types such as `int`).
- `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:

```
// A set of functions that move the cursor on a screen.
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();
```

Range-based `for` loops revisited

Range-based `for` loops

Traverse a `std::string`:

```cpp
bool is_all_digits(const std::string &str) {
  for (auto c : str)
    if (!std::isdigit(c))
      return false;
  return true;
}
```

Range-based `for` loops

Traverse a `std::vector`:

```cpp
bool is_all_digits(const std::string &str) {
  for (auto c : str)
    if (!std::isdigit(c))
      return false;
  return true;
}
int count_all_digits(const std::vector<std::string> &strs) {
  int cnt = 0;
  for (const auto &s : strs) // const std::string &s
    if (is_all_digits(s))
      ++cnt;
  return cnt;
}
```

Traverse an array

An array can also be traversed by range-`for`:

```
int arr[100] = {}; // OK in C++ and C23.
// The following loop will read 100 integers.
for (auto &x : arr) // int &
  std::cin >> x;
```

What else can be traversed using a range-`for`? $\Rightarrow$ We will learn about this when introducing **iterators**.

Pass an array by reference

```cpp
void print(int *arr) {
  for (auto x : arr) // Error: `arr` is a pointer, not an array.
    std::cout << x << ' ';
  std::cout << '\n';
}
```

We can declare `arr` to be a **reference to array**:

```cpp
void print(const int (&arr)[100]) {
  for (auto x : arr) // OK. `arr` is an array.
    std::cout << x << ' ';
  std::cout << '\n';
}
```

- `arr` is of type `const int (&)[100]`: a reference to an array of `100` elements, where each element is of type `const int`.

Pass an array by reference

We can declare `arr` to be a **reference to array**:

```
void print(const int (&arr)[100]) {
  for (auto x : arr) // OK. `arr` is an array.
    std::cout << x << ' ';
  std::cout << '\n';
}
```

- `arr` is of type `const int (&)[100]`: a reference to an array of `100` elements, where each element is of type `const int`.

Note that only arrays of `100` `int`s can fit here.

```
int a[100] = {}; print(a); // OK.
int b[101] = {}; print(b); // Error.
double c[100] = {}; print(c); // Error.
```

Pass an array by reference

To allow for arrays of any type, any length: Use a function template.

```
template <typename Type, std::size_t N>
void print(const Type (&arr)[N]) {
  for (const auto &x : arr)
    std::cout << x << ' ';
  std::cout << '\n';
}
```

We will learn about this in later lectures.

Summary

Type system

- Dangerous casts must happen explicitly: casting away low-level `const`ness, conversion between different pointer types, ...
- `const_cast`: used for casting away low-level `const`ness.
- `reinterpret_cast`: used for conversion between different pointer types.
- `static_cast`: used for some normal "innocent-looking" conversions: `int` to `double`, `unsigned` to `int`, ...
- **Prefer the C++-style named casts to old C-style casts.**
- `auto` and `decltype`: type deduction

Summary

Functions

- Default arguments: used for setting default values for some parameters.
- Function overloading: a group of functions have the same name, but different parameters so that they can be distinguished in the way they are called.

Range-based `for` loops

- Can also be used to traverse arrays.