CS100 Recitation 1

Dinghao Cheng

Special thanks: GKxx

Februrary 22, 2022

Useful website

- `https://i-techx.github.io/dashboard`
- Courses information in ShanghaiTech
- The website is hosted on Github Pages, so sometimes you may need VPN...

Contents

Language Standards

- Standards of C: C89/90, C99, C11, C17, C23 (coming soon).
- Standards of C++: C++98/03, C++11, C++14, C++17, C++20, C++23 (coming soon),...

Language Standards

- Standards of C: C89/90, C99, C11, C17, C23 (coming soon).
- Standards of C++: C++98/03, C++11, C++14, C++17, C++20, C++23 (coming soon),...
- A new version of standard C++ comes out every three years.

Language Standards

- Standards of C: C89/90, C99, C11, C17, C23 (coming soon).
- Standards of C++: C++98/03, C++11, C++14, C++17, C++20, C++23 (coming soon),...
- A new version of standard C++ comes out every three years.
- To specify a standard for the compiler, use `-std=c`*x* or `-std=c++`*y*, e.g. `-std=c11`, `-std=c++17`.

Language Standards

- Standards of C: C89/90, C99, C11, C17, C23 (coming soon).
- Standards of C++: C++98/03, C++11, C++14, C++17, C++20, C++23 (coming soon),...
- A new version of standard C++ comes out every three years.
- To specify a standard for the compiler, use `-std=c`*x* or `-std=c++`*y*, e.g. `-std=c11`, `-std=c++17`.
- To see what language standard the compiler is using, check the macro `__STDC_VERSION__` in C and `__cplusplus` in C++. For example, `__cplusplus == 201703L` means that the program is compiled under C++17.

Contents

Integer Types

- `short (int)`, `signed short (int)`, `unsigned short (int)`
- `int`, `signed int`, `unsigned int`
- `long (int)`, `signed long (int)`, `unsigned long (int)`
- `long long (int)`, `signed long long (int)`, `unsigned long long (int)` (since C99)

Integer Types

- What's the size of a `short`? `int`? `long`? `long long`?

Integer Types

- What's the size of a `short`? `int`? `long`? `long long`?
  `short` and `int` are at least 16-bit. `long` is at least 32-bit. `long long` is at least 64-bit.
  `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`

Integer Types

- What's the size of a `short`? `int`? `long`? `long long`?
  `short` and `int` are at least 16-bit. `long` is at least 32-bit. `long long` is at least 64-bit.
  `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
- Do `int` and `signed int` name the same type? What about others?

Integer Types

- What's the size of a `short`? `int`? `long`? `long long`?
  `short` and `int` are at least 16-bit. `long` is at least 32-bit. `long long` is at least 64-bit.
  `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
- Do `int` and `signed int` name the same type? What about others?
  For any integer type `T`, `T` and `signed T` name the same type.

Integer Types

Interesting fact

As with all the type specifiers, any order is permitted: `unsigned long long int` and `long int unsigned long` name the same type.

Integer Types

Interesting fact

As with all the type specifiers, any order is permitted: `unsigned long long int` and `long int unsigned long` name the same type.

- For the exact choices made by each implementation about the sizes of the integer types, you may refer to `https://en.cppreference.com/w/c/language/arithmetic_types`.

Integer Types

Interesting fact

As with all the type specifiers, any order is permitted: `unsigned long long int` and `long int unsigned long` name the same type.

- For the exact choices made by each implementation about the sizes of the integer types, you may refer to `https://en.cppreference.com/w/c/language/arithmetic_types`.
- Exact-width integer types like `int32_t` are defined in `stdint.h` since C99.

Boolean Type

The boolean type in C is **different** than that in C++.

- The type `bool` (same as `_Bool`) is defined since C99, in the header `stdbool.h`.

Boolean Type

The boolean type in C is **different** than that in C++.

- The type `bool` (same as `_Bool`) is defined since C99, in the header `stdbool.h`.
- Type `bool` holds two possible values: `true` and `false`.
- `true` and `false` are `#defined` as `1` and `0` respectively (until C23), so they have type `int` instead of `bool`. Since C23, their type will become `bool`.

Boolean Type

The boolean type in C is **different** than that in C++.

- The type `bool` (same as `_Bool`) is defined since C99, in the header `stdbool.h`.
- Type `bool` holds two possible values: `true` and `false`.
- `true` and `false` are `#defined` as `1` and `0` respectively (until C23), so they have type `int` instead of `bool`. Since C23, their type will become `bool`.
- How does the conversion between `bool` and integer types behave?

Boolean Type

The boolean type in C is **different** than that in C++.

- The type `bool` (same as `_Bool`) is defined since C99, in the header `stdbool.h`.
- Type `bool` holds two possible values: `true` and `false`.
- `true` and `false` are `#defined` as `1` and `0` respectively (until C23), so they have type `int` instead of `bool`. Since C23, their type will become `bool`.
- How does the conversion between `bool` and integer types behave?
  Nonzero $\Rightarrow$ `true`, zero $\Rightarrow$ `false`.
  `true` $\Rightarrow$ `1`, `false` $\Rightarrow$ `0`.

Character Types

- `char`, `signed char`, `unsigned char`
- Other types for wide characters: `wchar_t`, `char16_t`, `char32_t`.

Character Types

- `char`, `signed char`, `unsigned char`
- Other types for wide characters: `wchar_t`, `char16_t`, `char32_t`.
- Do `char` and `signed char` name the same type?

Character Types

- `char`, `signed char`, `unsigned char`
- Other types for wide characters: `wchar_t`, `char16_t`, `char32_t`.
- Do `char` and `signed char` name the same type?
  NO. The type `char` is neither `signed char` nor `unsigned char`. Whether `char` is signed depends on the implementation, but it is a **distinct type** (unlike the relationship between `int` and `signed int`).

Character Types

- `char`, `signed char`, `unsigned char`
- Other types for wide characters: `wchar_t`, `char16_t`, `char32_t`.
- Do `char` and `signed char` name the same type?
  NO. The type `char` is neither `signed char` nor `unsigned char`. Whether `char` is signed depends on the implementation, but it is a **distinct type** (unlike the relationship between `int` and `signed int`).
  To know the exact choices made by each implementation, see `https://en.cppreference.com/w/cpp/language/types`.

Character Types

- `char`, `signed char`, `unsigned char`
- Other types for wide characters: `wchar_t`, `char16_t`, `char32_t`.
- Do `char` and `signed char` name the same type?
  NO. The type `char` is neither `signed char` nor `unsigned char`. Whether `char` is signed depends on the implementation, but it is a **distinct type** (unlike the relationship between `int` and `signed int`).
  To know the exact choices made by each implementation, see `https://en.cppreference.com/w/cpp/language/types`.
- How do you save the returned value of `getchar`?

Character Types

- `char`, `signed char`, `unsigned char`
- Other types for wide characters: `wchar_t`, `char16_t`, `char32_t`.
- Do `char` and `signed char` name the same type?
  NO. The type `char` is neither `signed char` nor `unsigned char`. Whether `char` is signed depends on the implementation, but it is a **distinct type** (unlike the relationship between `int` and `signed int`).
  To know the exact choices made by each implementation, see `https://en.cppreference.com/w/cpp/language/types`.
- How do you save the returned value of `getchar`?
  `int` is recommended because `EOF` is `-1`.

Which Type to Use?

- Use `int` for integer arithmetic. `int` should be integer type that target processor works with most efficiently. If `int` is not large enough, use `long long`.
- Use `bool` for boolean values, especially in C++.
- Use **double** for floating-point computations.

Which Type to Use?

- Use `int` for integer arithmetic. `int` should be integer type that target processor works with most efficiently. If `int` is not large enough, use `long long`.
- Use `bool` for boolean values, especially in C++.
- Use **double** for floating-point computations.
  - The precision of `float` is usually not enough.
  - The cost of double-precision calculations versus single-precision is negligible. (In fact, double-precision operations are even faster on certain machines.)
  - The precision offered by `long double` is usually unnecessary.

Contents

Define a Function

- `return-type function-name(parameters) { function-body }`
- How to return a value?

Define a Function

- `return-type function-name(parameters) { function-body }`
- How to return a value?
  The `return` statement.

Define a Function

- `return-type function-name(parameters) { function-body }`
- How to return a value?
  The `return` statement.
- How to define a function without return-value?

Define a Function

- `return-type function-name(parameters) { function-body }`
- How to return a value?
  The `return` statement.
- How to define a function without return-value?
  Set the return-type to `void`.

Define a Function

- `return-type function-name(parameters) { function-body }`
- How to return a value?
  The `return` statement.
- How to define a function without return-value?
  Set the return-type to `void`.
- What happens when a function returns?

Define a Function

- `return-type function-name(parameters) { function-body }`
- How to return a value?
  The `return` statement.
- How to define a function without return-value?
  Set the return-type to `void`.
- What happens when a function returns?
  - The control flow goes back to the caller.
  - Possibly a value is passed to the caller.

Define a Function

Notice

Be sure to discriminate between the return of a function and the output of a program! They have nothing to do with each other.

Define a Function

Notice

Be sure to discriminate between the return of a function and the output of a program! They have nothing to do with each other.

Notice

A non-void function without a return statement causes no error (although probably a warning) when it is compiled, but results in undefined behavior when running!

The `main` Function

- You might have seen some people/textbooks writing 'void main'...

The `main` Function

- You might have seen some people/textbooks writing 'void main'...

  The definition 'void main' is not and has never been in C++, nor has it even been in C. (Bjarne Stroustrup)

The `main` Function

- You might have seen some people/textbooks writing 'void main'...

  The definition 'void main' is not and has never been in C++, nor has it even been in C. (Bjarne Stroustrup)

- You might have seen some people/textbooks leaving out the return-type...

The `main` Function

- You might have seen some people/textbooks writing 'void main'...

  The definition 'void main' is not and has never been in C++, nor has it even been in C. (Bjarne Stroustrup)

- You might have seen some people/textbooks leaving out the return-type...

  In C89, the default return-type of a function is `int`. However, this rule is not in standard C++ and has been dropped since C99. Don't be lazy!

The `main` Function

- You might have seen some people/textbooks writing 'void main'...

  The definition 'void main' is not and has never been in C++, nor has it even been in C. (Bjarne Stroustrup)

- You might have seen some people/textbooks leaving out the return-type...

  In C89, the default return-type of a function is `int`. However, this rule is not in standard C++ and has been dropped since C99. Don't be lazy!

- You might have seen many people leaving out the `return` statement in `main`...

The `main` Function

- You might have seen some people/textbooks writing 'void main'...

  The definition 'void main' is not and has never been in C++, nor has it even been in C. (Bjarne Stroustrup)

- You might have seen some people/textbooks leaving out the return-type...

  In C89, the default return-type of a function is `int`. However, this rule is not in standard C++ and has been dropped since C99. Don't be lazy!

- You might have seen many people leaving out the `return` statement in `main`...

  This is ok because the compiler will impose a return-value 0 if the program exits successfully.

Contents

Precedence and Associativity

- How is `a + b * c + d` evaluated?
- How is `a - b + c` evaluated?
- How is `f() + g() + h()` evaluated?

Precedence and Associativity

- How is `a + b * c + d` evaluated?
- How is `a - b + c` evaluated?
- How is `f() + g() + h()` evaluated?

Node

The precedence and associativity do not necessarily determine the evaluation order!

Precedence and Associativity

- How is `a + b * c + d` evaluated?
- How is `a - b + c` evaluated?
- How is `f() + g() + h()` evaluated?

Node

The precedence and associativity do not necessarily determine the evaluation order!

Typical undefined behavior: `printf("%d %d", a, ++a);`

Operator Precedence Table

Apart from the precedence of operators, you should also remember the associativities.

Table 4.4. Operator Precedence

Associativity and Operator	Function	Use	See Page
L ::	global scope	::name	286
L ::	class scope	class::name	88
L ::	namespace scope	namespace::name	82
L .	member selectors	object.member	23
L ->	member selectors	pointer->member	110
L []	subscript	expr [expr]	116
L ()	function call	name (expr_list)	23
L ()	type construction	type (expr_list)	164
R ++	postfix increment	lvalue++	147
R --	postfix decrement	lvalue--	147
R typeid	type ID	typeid(type)	826
R typeid	run-time type ID	typeid(expr)	826
R explicit cast	type conversion	*cast_name*<type>(expr)	162
R ++	prefix increment	++lvalue	147
R --	prefix decrement	--lvalue	147
R ~	bitwise NOT	~expr	152
R !	logical NOT	!expr	141
R -	unary minus	-expr	140
R +	unary plus	+expr	140
R *	dereference	*expr	53
R &	address-of	&lvalue	52
R ()	type conversion	(type) expr	164
R sizeof	size of object	sizeof expr	156
R sizeof	size of type	sizeof(type)	156
R sizeof...	size of parameter pack	sizeof...(name)	700
R new	allocate object	new type	458
R new[]	allocate array	new type[size]	458
R delete	deallocate object	delete expr	460
R delete[]	deallocate array	delete[] expr	460
R noexcept	can expr throw	noexcept (expr)	780

Short-circuit Evaluation

Logical operators && and || are short-circuited:

- Both && and || evaluates their left operand first.
- If the left operand of && evalutes `false`, the right operand will not be evaluated, and the whole expression evaluates `false`.
- If the left operand of || evalutes `true`, the right operand will not be evaluated, and the whole expression evaluates `true`.