CS100 Recitation 3

GKxx

March 7, 2022

Contents

How are the variables initialized?

```
int n, a[1000];

int main() {
  char b[20] = {1};
  double d;
  int i;
  i = 3;
  return 0;
}
```

- n: Value-initialized with 0.
- a: All elements value-initialized with 0.
- b: b[0] (explicitly) initialized with the character whose ASCII code is 1, others value-initialized with 0 (the null character).
- d: Default-initialized with an undefined value.
- i: Default-initialized with an undefined value.

Initialization vs Assignment

`int i = 0;` vs `int i; i = 0;`

- `int i = 0;` initializes the variable `i` with value `0`.
- `int i; i = 0;` default-initializes the variable `i`, and then assign `0` to it.
- The variable does not have a value before initialization, but has a value before assignment.
- Generally we prefer explicit initialization to assignment after declaration.

Contents

Name Lookup in C

- When referring to a name, only the names defined before are accessible.

```
int main() {
  int n; scanf("%d", &n);
  printf("%d\n", factorial(n)); // Error
  return 0;
}
int factorial(int n) {
  return n == 0 ? 1 : n * factorial(n - 1);
}
```

- Names will be checked from inner scopes to outer scopes. In other words, names in inner scopes hide those in outer scopes.

Example

```cpp
int i;
void fun() {
  int i = 42;
  // do something
}
int main() {
  int i = 0;
  for (int i = 0; i < 10; ++i) {
    // do something
  }
  for (int i = 0; i < 10; ++i)
    for (int i = 0; i < 100; ++i)
      ; // do something
  return 0;
}
```

Name Lookup before Type Checking

- During compilation, name lookup happens before type checking.
- That means, the difference in type cannot differentiate variables with the same name.

```
void fun() {
  // do something
}
int fun; // Error
```

Contents

Loops

```c
int n;
scanf("%d", &n);
while (n--) {
  // do something
}
```

- How many iterations are there?
  `n`.
- What's the value of `n` after execution?
  `-1`.
- Can we define `n` to be of type `unsigned`?

Loops

What about this?

```
for (unsigned i = n; i >= 0; --i) {
  // do something
}
```

The loop never ends, because an unsigned variable will never have a negative value!

Overflow and Underflow

- Overflow or underflow is not undefined behavior **only for unsigned integer types**.
- When an `n`-bit unsigned integer variable is assigned with a value x that is out of the representable range, it takes a nonnegative value that is less than 2^n and equivalent to x modulo 2^n.

Overflow and Underflow

提示

在你使用 C/C++ 的 `int` 类型时，如果发生了溢出，比较可能的情况是按照模 2^{32} 同余的前提下，在 `int` 范围内取一个合理的值。例如在计算 $2147483647+2$ 时，较有可能会得到 -2147483647。

然而，C/C++ 标准将这种情况归类为“未定义行为”。当你的程序试图计算会溢 出的 `int` 运算时，除了上述结果外，编译器还可能会让你的程序在此时计算出错误结果、死循环、运行错误等，这也是符合 C/C++ 标准的。

如果你的程序希望利用 `int` 的自然溢出的特性，请转换为 `unsigned` 类型运算。例如将 `a + b` 改写为 `(int) ((unsigned) a + (unsigned) b)`，以避免出现不预期的错误。

Infinite Loops

`while (true)` + `break` can be used as substitute for `do-while` loops.

```
// Why is n declared
   here?
int n;
do {
  scanf("%d", &n);
  if (n < 0)
    printf("Please input
        again!\n");
} while (n < 0);
```

```
while (true) {
  int n;
  scanf("%d", &n);
  if (n < 0)
    printf("Please input
        again!\n");
  else
    break;
}
```

break vs continue

Explain the behavior of the following code.

```
for (int i = 0; i < n; ++i) {
  if (a[i] % 2 == 1)
    continue;
  int x = calc(a[i]);
  if (check(x))
    break;
  update(a[i]);
  ++count;
}
```

break vs continue

Explain the behavior of the following code.

```
for (int i = 0; i < n; ++i) {
  if (a[i] % 2 == 1)
    continue;
  int x = calc(a[i]);
  if (check(x))
    break;
  update(a[i]);
  ++count;
  // 'continue' goes here.
}
// 'break' goes here.
```

Variable Declaration in `switch-case`

Due to the special control path of `switch-case` statements, any case branch that contains a variable declaration must be a block.

```
switch (a) {
  case 1: {
    int x = calc(a);
    // do something
    break;
  }
  case 2:
    // x cannot be used here.
    break;
  default:
    break;
}
```

Contents

Constant Expressions

- Constant expressions refer to the expressions that can be evaluated during compile-time.
- In C:
  - Expressions that only contain literals
  - `enum` hack
- `#define PI 3.14`

  Is `PI` a constant expression?

  Yes, because `PI` will be replaced by the literal `3.14`.
- `const int maxn = 100;`

  Is `maxn` a constant expression?

  No. `maxn` is a constant variable.

Constant Expressions

The value of `const` variables cannot be changed after initialization, but may not be determined during compile time.

```
int i;
scanf("%d", &i);
const int j = i;
```

- `const` variables initialized with a constant expression is also constant expression in C++, but not in C.
- `const int maxn = 100;` is a constant expression in C++, but not in C.

enum Hack

```c
enum { maxn = 100 };
int a[maxn];
```

- `maxn` has type `int`, and it is a constant expression.
- Use `enum` hack to define `bool`:

  ```c
  typedef enum { false, true } bool;
  ```

⇒ *Effective C++*, Item 2.

Define an Array

`type name[N];`

- `N` **must be a constant expression**. (We will talk about this later.)
- The following code is illegal before C99, even though many compilers are so smart that they can handle it.

```
const int maxn = 1000;
int a[maxn]; // Error: maxn is not a constant
    expression.
```

Element Access

- Access through subscript: `a[i]`.
- `*(a + i)`: an equivalent way, but treats array as a pointer.
- In fact, subscript operator also works on pointers: `p[i]` is the same as `*(p + i)` for a pointer `p`.
- What does `scanf("%d", a + i)` mean?
  Same as `scanf("%d", &a[i])`.

Traversal

- Through subscript:

```
for (int i = 0; i < n; ++i)
  do_something(a[i]);
```

- Through pointer:

```
for (int *p = a, *end = a + n; p != end; ++p)
  do_something(*p);
```

- More fancy way:

```
int *p = a, *end = a + n;
while (p != end)
  do_something(*p++);
```

The '*' Specifier

Use '*' to define a pointer.

- Both `int *p` and `int* p` are right,
- but the latter may fool you in some cases:

```
int* p1, p2, p3;
```

- Choose one way and persist. If you choose `int* p`, never define more than one pointers in one declaration!

Confusing Types

- `int (*a)[10]`: `a` is a pointer, which points to an array, which stores 10 `int`s.
- `int *a[10]`: `a` is an array, which stores 10 pointers, each pointing to an `int`.
- Use type alias:

```
typedef int arr_t[10];
arr_t *a;
typedef int *ptr_t;
ptr_t pa[10];
```

Constant Types

- `const int *p` and `int const *p` are the same: a pointer, which points to a `const int`.
- `int *const p`: a constant variable, which is a pointer, which points to an `int`.
- When a variable itself is constant, it is a top-level const. When a variable is a pointer that points to a constant variable, it is a low-level const.
- `const int *const p` is both top-level const and low-level const.

`const` and Pointers

- Low-level const pointers can point to non-const variables, which is called '**adding low-level const**'.

```
int i = 42;
const int *p = &i;
```

- Modifying `i` through `p` is not allowed, but it can be modified in other ways.
- **Deleting low-level const** is not allowed:

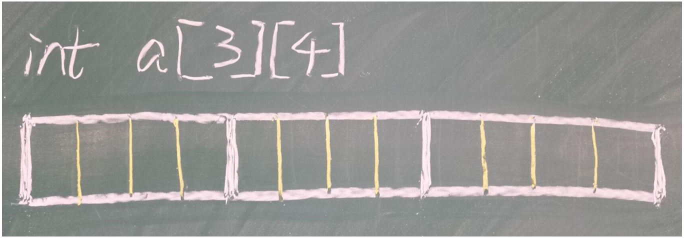
```
int *p2 = p; // Error
```

⇒ *Effective C++*, Item 3.

Multi-dimensional Arrays

There's no so-called multi-dimensional arrays in C/C++. Instead, they are arrays of arrays.

- `int a[3][4];` 'a' is an array of size 3, where each element is an array of 4 `int`s.
- Storage of 2d-array: Not a matrix!



`size_t` and `ptrdiff_t`

Defined in header `stddef.h`.

- `size_t` is an unsigned integer type of the result of `sizeof`.
- `size_t` can store the maximum size of a theoretically possible object of any type.
- `ptrdiff_t` is a signed integer type of the result of subtracting two pointers.
- Both `size_t` and `ptrdiff_t` are implementation-defined.

Variable Length Array

- Since C99, the length of arrays is allowed to be determined during runtime.
- Since C11, compilers may define the macro `__STDC_NO_VLA__` to integer `1` to indicate that VLA is not supported.
- VLA is constructed on stack, while the 'dynamic-arrays' allocated by `malloc` are on heap.
- **VLA has never been supported by standard C++.**
- We do not recommend to use VLA. Instead, use dynamic memory when the length of array is determined during runtime.
- https://en.cppreference.com/w/c/language/array

Contents

Declaration vs Definition

- Declaration without definition:
  `return-type function-name(params);`
- Definition:
  `return-type function-name(params) {function-body}`
  There is no semicolon at the end of function definition!
- A function can be declared any times, but only defined once.
- A definition is also a declaration.
- There should be at least one declaration of the function before it is called.
- In a declaration, the names of the parameters can be omitted, as they are not used.

Calling a Function

```c
int factorial(int n) {
  int s = 1;
  for (int i = 1; i <= n; ++i)
    s *= i;
  return s;
}
int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", factorial(n));
  return 0;
}
```

Calling a Function

```
int result = factorial(n);
```

- The '()' is called the function-call operator.
- The function-call operator cannot be omitted, even if the function takes no arguments.
- Statements that do nothing:

```
;
5;
2 + 3;
{}
n;
fun;
```

Passing Arrays to Functions

Define an array parameter:

- `int *a`, `int a[]` and `int a[n]` are **totally the same**: array types decay to pointer types.
- C functions have no way of knowing the length of an array parameter.
- C functions cannot require the array parameter to be of any certain length. (They cannot even require it to be an array!)
- The following code compiles, but may cause disaster.

```
void fun(int a[10]) {}
int i;
fun(&i);
```

Passing Arrays to Functions

Example:

```
void print_array(int *a, int n) {
  for (int i = 0; i < n; ++i)
    printf("%d ", a[i]);
}
int main() {
  int arr[] = {2, 5, 6};
  print_array(arr, 3);
  return 0;
}
```

Passing Arrays to Functions

```c
void print_array2(int *begin, int *end) {
  for (int *p = begin; p != end; ++p)
    printf("%d ", *p);
}
void print_array3(int *begin, int *end) {
  while (begin != end)
    printf("%d ", *begin++);
}
int main() {
  int arr[] = {2, 5, 6};
  print_array2(arr, arr + 3);
  print_array3(arr, arr + 3);
  return 0;
}
```

Passing Multi-dimensional Arrays to Functions

- What type will `int [3][4]` decay to?
  `int [3][4]` is an array of `int [4]`, so it will decay to a pointer that points to `int [4]`, that is:
  `int (*)[4]`
- Differentiating `int (*a)[4]` and `int *a[4]`.
  `int (*a)[4]` is a pointer that points to `int [4]`,
  while `int *a[4]` is an array of four pointers, each pointing to an `int`.

Passing Muldi-dimensional Arrays to Functions

```
void print_2darray(int (*a)[4], int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < 4; ++j)
      printf("%d ", a[i][j]);
    puts("");
  }
}
int main() {
  int a[3][4] = /* some value */;
  print_2darray(a, 3);
  return 0;
}
```

The size '4' cannot be left out, otherwise it will become an incomplete type (`int (*)[]`).

Safety Issue of scanf

Use `scanf` to read a string:

```
char str[100];
scanf("%s", str);
```

- `scanf` has no idea how big your array is!
- **Array subscript out of range** is severe runtime error, which cannot be detected during compile-time, and may not report when happening. (On Linux systems, it reports a 'segmentation fault'.)
- Functions like `gets` are removed in modern C and C++ due to similar issues. Functions like `scanf_s` are introduced for safety.

Modifying Outer Variables

The following definition of a 'swap' function does not work:

```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

Because `a` and `b` are local variables of the function.
When `swap(x, y)` is called, the variables `a` and `b` are initialized with values of `x` and `y` respectively. In other words, they are copies of `x` and `y`.

Modifying Outer Variables

Pass by pointer:

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Question

Why is '&' needed when passing a variable to `scanf`, but not needed for `printf`?

Returning Multiple Values

As in HW2-1:

```
void findSecondMaxAndMin(int a[], int size, int
    *secondMin, int *secondMax) {
  *secondMin = /* some value */;
  *secondMax = /* some value */;
}
```

Can we write as in Python?

```
return (secondMin, secondMax);
```

The Comma Operator

- The comma in the expression '`a, b`' is the comma operator. It is the operator of the lowest precedence.
- The evaluation order is determined! (4)
- The left operand is evaluated first, and then the right operand is evaluated.
- The return value of the comma expression is the value of the right operand.

```
// a is initialized with value of c.
int a = (b, c);
```

- Not all commas are comma operators. Some work as a part of the grammar.

Function Inlining

```
#define MAX(A, B) ((A) < (B) ? (B) : (A))
```

Pros and cons?

- Time- and memory-saving, in comparison with functions.
- May cause unexpected results:

  ```
  int x = MAX(++i, j);
  ```

- We want the compiler to expand the function at the call site instead of calling it, so that the time and memory cost could be reduced.

Function Inlining

```
inline double max(double a, double b) {
  return a < b ? b : a;
}
```

- The `inline` specifier is a hint for the compiler to perform inline expansion.
- Compilers have the right to accept or ignore the `inline` specifier.
- Usually, inline request will be accepted for simple and short functions,
- and ignored for functions that are too long or recursive.
- Function inlining are not without drawbacks.

⇒ *Effective C++*, Item 30.