

## CS100 Recitation 4

GKxx

March 7, 2022

# Contents

- 1 Pointers and Arrays
  - Basic Knowledge
  - Dynamic Memory
- 2 C-style Strings
- 3 Type Casting
  - Safe Conversions
  - Dangerous Conversions
  - Summary
- 4 A Peek of C++
  - Type Casting
  - Function Overloading

## '\*' and '&'

### 解引用



- In declaration statements, '\*' is the **pointer specifier**.
- In an expression, '\*' is the **dereference operator**.
- For a pointer p, \*p is **dereferencing the pointer**, which returns the **object** that p points to.
- In an expression, '&' is the **address-of operator**, which takes the address of the operand.

# Conversion from Array to Pointer

```
int a[10];  
int *p1 = a;  
int *p2 = &a[0];
```

- The array can be **implicitly converted** to the address of the first element,
- **but pointer and array are different types!**

# Conversion from Array to Pointer

```
int a[10];  
int *p1 = a;  
int *p2 = &a[0];
```

- The array can be **implicitly converted** to the address of the first element,
- **but pointer and array are different types!**
- $p1 + 3$  is the same as  $\&p1[3]$ .

## Theorem

Suppose  $p$  is defined to be a pointer of type  $T *$  and  $i$  is an integer. Then  $(p + i)$  is the address obtained by shifting from  $p$  by  $i * \text{sizeof}(T)$  bytes.

## Theorem

Suppose  $p$  is defined to be a pointer of type  $T *$  and  $i$  is an integer. Then  $(p + i)$  is the address obtained by shifting from  $p$  by  $i * \text{sizeof}(T)$  bytes.

```
int a[10];  
int *pi = a;  
int (*pa)[10] = &a;  
printf("%p, %p\n", pi, pa);  
printf("%p, %p\n", pi + 3, pa + 3);
```

# Value of Pointer

At any time, a pointer might be:

- 1 pointing to an object.
- 2 pointing to the location just immediately past the end of an object.
- 3 **NULL**, indicating that it is not pointing to any object.
- 4 pointing to nowhere (**invalid**); values other than the preceding three are invalid.



# Value of Pointer

- A pointer has **invalid** value after **default initialization**.
- A pointer is **NULL** after **value initialization**.

# Value of Pointer

- A pointer has **invalid** value after **default initialization**.
- A pointer is **NULL** after **value initialization**.
- Default-initialized pointers that point to nowhere are called **wild pointers**.
- Pointers also have **invalid** value after being freed. This is called a **dangling pointer**.

# Value of Pointer

## Theorem

A pointer is **dereferencable** if and only if it is pointing to an object.

- Taking an invalid address as the value of a pointer is okay,
- but any attempt to dereference an **undereferencable** pointer is severe runtime-error.

# Value of Pointer

## Theorem

A pointer is **dereferencable** if and only if it is pointing to an object.

- Taking an invalid address as the value of a pointer is okay,
- but any attempt to dereference an **undereferencable** pointer is severe runtime-error.

- `int a[10];`  
`int (*p)[10] = &a;`

`p + 3` is okay, but `*(p + 3)` and `p[3]` cause runtime-error!

# Magic Square

```
int **magicSquare(int n) {  
    int **p = malloc(sizeof(int) * n);  
    for (int i = 0; i < n; ++i)  
        p[i] = malloc(sizeof(int) * n);  
    // What is the value of each p[i][j] now?  
    /* Fill the magic square. */  
    return p;  
}  
  
void freeMagicSquare(int **p, int n) {  
    for (int i = 0; i < n; ++i)  
        free(p[i]);  
    free(p);  
}
```

# Magic Square

How many problems are there?

```
int **magicSquare(int n) {  
    int p[n][n];  
    // Fill the magic square.  
    return p;  
}
```

# Magic Square

How many problems are there?

```
int **magicSquare(int n) {  
    int p[n][n];  
    // Fill the magic square.  
    return p;  
}
```

- VLA is not recommended in C and forbidden in C++.
- `p` is a **local variable** of the function, which is destroyed immediately the function ends.
- `int [n][n]` can be converted to `int (*)[n]` naturally, but then the conversion from `int (*)[n]` to `int **` is severe runtime-error! (We will talk about this later.)

# Usage of free

`free(p)` frees the memory that starts at the address pointed by `p`.

- If `p` is not pointing to some memory that is dynamically allocated, `free(p)` causes runtime-error.
- The system knows the size of the memory, and the entire piece of memory will be freed. You cannot free only a part of it.
- Any attempt to free only a part of a piece of memory causes runtime-error.



# Contents

- 1 Pointers and Arrays
  - Basic Knowledge
  - Dynamic Memory
- 2 C-style Strings
- 3 Type Casting
  - Safe Conversions
  - Dangerous Conversions
  - Summary
- 4 A Peek of C++
  - Type Casting
  - Function Overloading

# Definition

## Definition

A group of characters stored in contiguous memory terminated with a null character `'\0'` is a C-style string.

# Definition

## Definition

A group of characters stored in contiguous memory terminated with a null character `'\0'` is a C-style string.

- The value of a `char` variable is exactly the ASCII of the character it represents.
- ASCII of `'\0'`: 0.
- `char str[6] = "Hello";`
- The **length** of a string is the number of real characters in the string, in which the null character is not counted.

# The `string.h` Library

Make sure you know them before using!

- `strlen(s)` returns the length of the string `s`. The returned value is of type `size_t`.

# The string.h Library

Make sure you know them before using!

- `strlen(s)` returns the length of the string `s`. The returned value is of type `size_t`.
- `strcmp(s1, s2)` compares `s1` and `s2` in lexicographical order, returns
  - a positive value if `s1 > s2`.
  - 0 if `s1 == s2`.
  - a negative value if `s1 < s2`.

词典编纂的

It is not 1, 0 or -1!

# The string.h Library

Make sure you know them before using!

- `strlen(s)` returns the length of the string `s`. The returned value is of type `size_t`.
- `strcmp(s1, s2)` compares `s1` and `s2` in lexicographical order, returns
  - a positive value if `s1 > s2`.
  - 0 if `s1 == s2`.
  - a negative value if `s1 < s2`.

It is not 1, 0 or -1!

- Make sure that the string you pass to functions in the standard library (including IO functions) is null-terminated, otherwise it is undefined behavior.

# Usage of strlen

strlen counts the characters by traversing the whole string until reaching the null character.

- The following code is very slow:

```
for (size_t i = 0; i < strlen(s); ++i)  
    // do something with s[i]
```

## Usage of strlen

strlen counts the characters by traversing the whole string until reaching the null character.

- The following code is very slow:

```
for (size_t i = 0; i < strlen(s); ++i)
    // do something with s[i]
```

- Correct way:

```
size_t len = strlen(s);
for (size_t i = 0; i < len; ++i)
    // do something with s[i]
```



# String Literals

- String literals are of type `char [N + 1]`, where N is the length of the string.
  - In C++, string literals are of type `const char [N + 1]`, which is more reasonable.

# String Literals

- String literals are of type `char [N + 1]`, where N is the length of the string.
  - In C++, string literals are of type `const char [N + 1]`, which is more reasonable.
- Unlike literals of other types, `string literals` are stored in static storage and **cannot be modified**.
  - You can take the address of a string literal:  

```
printf("%p\n", &"Hello");
```

# String Literals

- String literals are of type `char [N + 1]`, where N is the length of the string.
  - In C++, string literals are of type `const char [N + 1]`, which is more reasonable.
- Unlike literals of other types, **string literals** are stored in static storage and **cannot be modified**.
  - You can take the address of a string literal:  
`printf("%p\n", &"Hello");`
  - `char *ptr = "Hello";` is defining a pointer pointing to that literal.
  - `char str[] = "Hello";` is defining an array and copying the values from that literal.

# String Literals

- String literals in C have a non-const type, but have a **const semantic**. 语义
- The following is undefined behavior:

```
char *ptr = "Hello";  
ptr[2] = 'B';
```

while this is okay:

```
char str[] = "Hello";  
str[2] = 'B';
```

# String Literals

- String literals in C have a non-const type, but have a **const semantic**.
- The following is undefined behavior:

```
char *ptr = "Hello";  
ptr[2] = 'B';
```

while this is okay:

```
char str[] = "Hello";  
str[2] = 'B';
```

- It is recommended to use a low-level const pointer to point to a string literal, as in C++.

# Pitfall of C-style Strings

Consider a function that returns a **slice** (substring) of a string.

```
inline char *strslice
    (char const *str, size_t l, size_t r) {
    char result = ???
    for (size_t i = l; i < r; ++i)
        result[i - l] = str[i];
    return result;
}
```

How do you store the result?

# Pitfall of C-style Strings

How do you store the result?

# Pitfall of C-style Strings

How do you store the result?

- Array/VLA is not allowed because it is a local variable that cannot be **returned**.



# Pitfall of C-style Strings

How do you store the result?

- Array/VLA is not allowed because it is a local variable that cannot be **returned**.
- **static** array won't be destroyed until the program terminates, but VLA cannot be **static**, so ...

```
static char result[10000];
```

- The size '10000' is quite problematic.

# Pitfall of C-style Strings

How do you store the result?

- Array/VLA is not allowed because it is a local variable that cannot be **returned**.
- **static** array won't be destroyed until the program terminates, but VLA cannot be **static**, so ...

```
static char result[10000];
```

- The size '10000' is quite problematic.
- What's worse, it **confuses the user**. Each call to this function affects the result of previous ones.

## Pitfall of C-style Strings

The only reasonable choice seems to be ...

```
char *result = malloc(sizeof(char) * (r - 1));
```

## Pitfall of C-style Strings

The only reasonable choice seems to be ...

```
char *result = malloc(sizeof(char) * (r - 1));
```

However, this causes a lot of trouble for the user.

- The user needs to know the implementation detail that the memory is dynamically allocated.
- The user needs to make sure that the memory is correctly freed.

# Pitfall of C-style Strings

## 缺陷

The only reasonable choice seems to be ...

```
char *result = malloc(sizeof(char) * (r - 1));
```

However, this causes a lot of trouble for the user.

- The user needs to know the implementation detail that the memory is dynamically allocated.
- The user needs to make sure that the memory is correctly freed.

The task of memory management is done by **both the user and the designer!**

# Pitfall of C-style Strings

How do the standard libraries deal with this?

# Pitfall of C-style Strings

How do the standard libraries deal with this?

```
char *strcpy(char *dest, const char *source);  
char *strcat(char *dest, const char *source);
```

They always require users to allocate space for the results!

# Pitfall of C-style Strings

How do the standard libraries deal with this?

```
char *strcpy(char *dest, const char *source);  
char *strcat(char *dest, const char *source);
```

They always require users to allocate space for the results!

- In this way, memory management is done only by the user.  
But is that good enough?
- Who is it that should take on this task?



# Pitfall of C-style Strings

- When we do things with a C-style string, we are in fact doing things with the memory directly.
- Is there a kind of 'string' that manages its own memory on itself?

## Pitfall of C-style Strings

- When we do things with a C-style string, we are in fact doing things with the memory directly.
- Is there a kind of 'string' that manages its own memory on itself?
- Yes, but in C++.

⇒ *Ruminations on C++, Part 1 Chapter 1: Why I use C++.*

**Theorem (Fundamental Theorem of Software Engineering, FTSE)**

We can solve any problem by introducing an extra level of indirection.

Said Butler Lampson, referred to as 'FTSE' by Andrew Koenig.

# Contents

- 1 Pointers and Arrays
  - Basic Knowledge
  - Dynamic Memory
- 2 C-style Strings
- 3 Type Casting**
  - Safe Conversions
  - Dangerous Conversions
  - Summary
- 4 A Peek of C++
  - Type Casting
  - Function Overloading

# Implicit and Explicit Casting

- Implicit casting: happen automatically.
- Explicit casting: done manually.

# Implicit and Explicit Casting

- Implicit casting: happen automatically.
- Explicit casting: done manually.

```
int a = some_value(), b = some_value();  
double average = (a + b) / 2.0;  
double average2 = (double)(a + b) / 2;
```

# Implicit and Explicit Casting

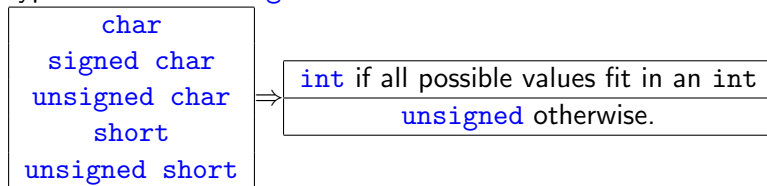
- Implicit casting: happen automatically.
- Explicit casting: done manually.

```
int a = some_value(), b = some_value();
double average = (a + b) / 2.0;
double average2 = (double)(a + b) / 2;
```

- In C, we use (T)expr to explicitly convert the value of expr to type T.
- However, some conversions are very dangerous!

# Integral Promotion

Integral promotion refers to the conversion from small integer types to `int` or `unsigned int`.



# Arithmetic Conversion

Apart from integral promotion, all other types of conversion between arithmetic types are called **arithmetic conversion**.

- The conversion between integer types and floating-point types.
- The conversion between character types and floating-point types.
- The conversion between signed and unsigned types.

Refer to *C++ Primer* Chapter 4 Section 4.11, or  
<https://en.cppreference.com/w/c/language/conversion>.



# Common Type

When the operands of arithmetic or relationship operators are of different types, they will be converted to a common type.

# Common Type

When the operands of arithmetic or relationship operators are of different types, they will be converted to a common type.

- The rules for integral promotion and arithmetic conversion are very complicated, but several cases are common to see:

```
int i = -1;
const char *str = "Hello";
if (i < strlen(str)) // A warning here.
    puts(str);
```

# Common Type

When the operands of arithmetic or relationship operators are of different types, they will be converted to a common type.

- The rules for integral promotion and arithmetic conversion are very complicated, but several cases are common to see:

```
int i = -1;
const char *str = "Hello";
if (i < strlen(str)) // A warning here.
    puts(str);
```

- Integer types, if necessary, will always be converted to floating-point types.

```
float fval = 3.14;
long long llval = 998244353;
// The type of fval * llval is float.
```

## const Conversions

```
int i = 42;
// Adding low-level const
const int *cip = &i;
// Adding top-level const
const int *const cicp = cip;
// Removing top-level const
const int *cip2 = cicp;
// Dangerous: removing low-level const
int *ip = cip;
```

## const Conversions

```
int i = 42;
// Adding low-level const
const int *cip = &i;
// Adding top-level const
const int *const cicp = cip;
// Removing top-level const
const int *cip2 = cicp;
// Dangerous: removing low-level const
int *ip = cip;
```

Adding low/top-level `const` or removing top-level `const` are safe,  
but removing low-level `const` is dangerous!

## const Conversions

Unless the source pointer is really pointing to a non-const object, removing low-level `const` is **undefined behavior**.

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

## const Conversions

Unless the source pointer is really pointing to a non-const object, removing low-level `const` is **undefined behavior**.

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

If you really need to cast away low-level `const`:

- Make sure the source pointer is pointing to a non-const object.
- Do it **explicitly**. (In fact, such conversion cannot happen implicitly in C++.)

# Conversions between Pointers

```
int i = 42;  
double *dp = &i; // Very dangerous.
```

- It is in fact **reinterpreting** the memory pointed by the source pointer.
- Avoid such conversion unless you really know what you are doing!
- Do it **explicitly** if it is really needed.



# Conversions between Pointers

```
void fun(int **a) {  
    // do something  
}  
int arr[3][4] = {0};  
fun(arr);
```

This is converting `int (*)[4]` (decayed from `int [3][4]`) to `int **`, which is undefined behavior.

# Conversions between Pointers

The `void *` type:

- Any pointer can be converted to `void *`. (safe)
- You can obtain nothing by dereferencing a `void *`. (forbidden in C++)

# Conversions between Pointers

The `void *` type:

- Any pointer can be converted to `void *`. (safe)
- You can obtain nothing by dereferencing a `void *`. (forbidden in C++)
- A `void *` can be converted to any pointer type,
- but it is **undefined behavior** unless the conversion preserves the real type of the pointer.

# Conversions between Pointers

The `void *` type:

- Any pointer can be converted to `void *`. (safe)
- You can obtain nothing by dereferencing a `void *`. (forbidden in C++)
- A `void *` can be converted to any pointer type,
- but it is **undefined behavior** unless the conversion preserves the real type of the pointer.

## Question

How does `scanf` work?

# Conversions between Pointers

```
double dval = 3.14;  
printf("%d\n", dval);  
scanf("%d", &dval);
```

# Conversions between Pointers

```
double dval = 3.14;  
printf("%d\n", dval);  
scanf("%d", &dval);
```

- printf: conversion from `double` to `int`, which is safe, though may lose precision.
- scanf: conversion from `double *` to `int *`, which is **undefined behavior**.

# Conversions between Pointers

```
double dval = 3.14;  
printf("%d\n", dval);  
scanf("%d", &dval);
```

- `printf`: conversion from `double` to `int`, which is safe, though may lose precision.
- `scanf`: conversion from `double *` to `int *`, which is **undefined behavior**.
- `scanf` has no idea what types of pointers you pass to it, so it first uses `void *` for every pointer, and then converts them according to the format string.

## Conversion between Pointers and Integers

```
int i = 42;  
int j = &i;  
printf("%d\n", j);
```

- This will assign the address of `i` to `j` as an integer value, and then print it in decimal.



# Conversion between Pointers and Integers

```
int i = 42;  
int j = &i;  
printf("%d\n", j);
```

- This will assign the address of `i` to `j` as an integer value, and then print it in decimal. 十进制
- **AVOID such conversion!**
  - Its behavior is implementation-defined.
  - It may have other issues...
  - It is forbidden in C++.

# Conversion between Pointers and Integers

```
int i = 42;  
int j = &i;  
printf("%d\n", j);
```

- This will assign the address of `i` to `j` as an integer value, and then print it in decimal.
- **AVOID such conversion!**
  - Its behavior is implementation-defined.
  - It may have other issues...
  - It is forbidden in C++.
- If you want to see the difference between two pointers, use **subtraction!**

# Summary

## Safe

Top-level `const` conversion  
Decay of arrays (functions?)  
Adding low-level `const`  
Integer promotion  
Arithmetic conversion

## Dangerous

Casting-away low-level `const`  
Reinterpreting pointers  
Conversion from `void *`

- Remember and distinguish between different kinds. (This is rather important in C++.)
- “Dangerous” type casting can happen implicitly without error in C, **but recognize them and do them explicitly!**
- “Dangerous” type casting must be carried out explicitly in C++.

# Contents

- 1 Pointers and Arrays
  - Basic Knowledge
  - Dynamic Memory
- 2 C-style Strings
- 3 Type Casting
  - Safe Conversions
  - Dangerous Conversions
  - Summary
- 4 A Peek of C++
  - Type Casting
  - Function Overloading

# Named Type-casting Operators

C++ has four **named type-casting operators**:

- **static\_cast**: for “safe” type-casting and conversion from **void \***.
- **const\_cast**: for casting away low-level **const**.
- **reinterpret\_cast**: for pointer conversion.
- **dynamic\_cast**: for runtime polymorphic downcasting.

# Named Type-casting Operators

C++ has four **named type-casting operators**:

- **static\_cast**: for “safe” type-casting and conversion from **void \***.
- **const\_cast**: for casting away low-level **const**.
- **reinterpret\_cast**: for pointer conversion.
- **dynamic\_cast**: for runtime polymorphic downcasting.

Usage: `cast-name<type>(expr)`.

# Named Type-casting Operators

```
int a = 42, b = 57;  
double average = static_cast<double>(a + b) / 2;  
const int *cip = &a;  
// Is this const_cast safe? Why?  
int *ip = const_cast<int *>(cip);  
char *cp = reinterpret_cast<char *>(ip);
```

# Named Type-casting Operators

```
int a = 42, b = 57;  
double average = static_cast<double>(a + b) / 2;  
const int *cip = &a;  
// Is this const_cast safe? Why?  
int *ip = const_cast<int *>(cip);  
char *cp = reinterpret_cast<char *>(ip);
```

## Question

Benefits of these type-casting operators?



# Named Type-casting Operators

```
int a = 42, b = 57;
double average = static_cast<double>(a + b) / 2;
const int *cip = &a;
// Is this const_cast safe? Why?
int *ip = const_cast<int *>(cip);
char *cp = reinterpret_cast<char *>(ip);
```

## Question

Benefits of these type-casting operators?

⇒ *Effective C++*, Item 27.

# Overloaded Functions

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

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

# Match of Overloaded Functions

Suppose we have the following overloaded functions

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

How does a function call find the best match?

# Match of Overloaded Functions

- 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 promotion.
- ④ Match through arithmetic conversion.
- ⑤ Match through a class-type conversion.