Compiling C++ Code

Stages of compilation

pre-processor stage: expanding macro, include files(files path from compiler default, the environment, or, command line –I option)

compilation stage: main stage, in which Code transformed into an object code.

linker: link object files of program to other object files and libraries.

inline expansion

template instantiation

code analysis, optimization & generation

```
.C .cc .cpp          source file
.i                   pre-processor output
.s .S                assembler source
.o                   object file
.a                   static library for linker
.so                  dynamic library for linker
```

Standard Compiler options

```
g++ [options] [source-files] [object-files] [libraries]
```

```
-V            print compiler version number
-c            compile only, producing a .o for linking
-o            set name of output executable
-g            prepare file for debugging
-O?           Control optimiser, -O3 highest
-l            add library to linkers list
-L<path>      add path to the library search path
-I<path>      add path to pre-processor include path
```

Type and Variables

C++ all variables must have a type.

type 可以使 compiler 知道该 reserve and structured how much memory, 可表明该变量适用哪些 operations.

types are referred to as classes

variables are referred to as objects

OOsystem class 不仅联系 object 且关联适用哪些 operations

Fundamental Types

To represent integers of different sizes

	bytes
bool	1
char	1
wchar_t	2
short int	2
int	4
long int	4
long long	8

To represent floating point numbers

float	4
double	8
long double	16

Namespace

package classes, objects & functions within a single scope

avoid name collisions

Literal constants

are actual values and are typed

handle by compiler, when program is executed, placed in memory

```
200          // decimal
023          // octal
0xff         // hexadecimal
```

int i;

i = 33;

there is a period during the program in which the values were unknown → dangerous state

Variables may be initialized when they are declared.

```
double d = 1.2;
float f = 3.2f;
unsigned long long u = 10ull;
int b = 0b00101010;
auto m = 1'000'000;

using namespace std::chrono_literals;
auto h = 10h;
```

Auto Deduction

auto 可 deduce type, 之后该变量不能改 type

deduction works out type needed to hold the value of the initializer.

Types of Initialization

1. List initialization

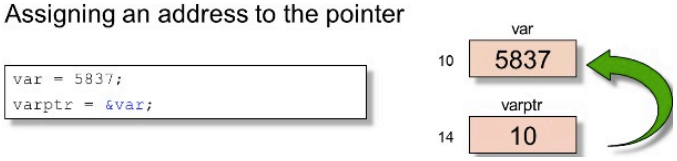
   {}
   禁止 implicit narrowing conversions
   int x{42.3} ❌ →会 compiler error

2. Copy initialization (inherited from C)
   = //not assignment, its copy init
   允许 implicit narrowing conversions
   int x = 42.3 → works, compiler warning
3. Default initialization
   int i; double d; 会是 indeterminate values
   属于 Undefined Behavior
   may 用在 we assign a value to it at runtime.
4. Value initialization
   int j{ }; init with default values. (0)

Pointer type

A pointer is a variable that may contain the address of a variable

Assigning an address to the pointer



允许 multiple references to one memory location.

eg: int *p = &var, *q = &var …

Important operators for pointers

*	indirection operator, allows access through pointer
&	address operator, takes the address of a variable

integer pointer may not hold the address of a float variable. 可 hold，但 不建议

nullptr 指 the pointer doesn't point anywhere.

Reference type

Automatic pointer mechanism

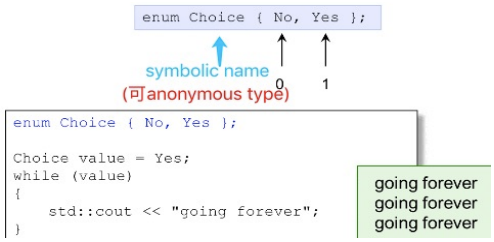
- variable 别名 aliases
- less flexible than pointer, but no longer required & and *

int i = 5; **int &ir;** ir = i; ❌ Reference must be initialized.

int i = 5; int &ir = i; **++i, ++ir** 本质操作都一样

Enumeration type

- user-defined set of values, represent integral values
- exist only at compile time (不占 run time,快)
- meaningful and convenient



it is impossible to make variables specifically of the enumeration type.

Light t = RED;

✅ t = 22

int x = 10;

❌ x = RED;

Array

int a[10];

cout << a[10]; java 不会 error at compile time

c++ danger, Undefined Behavior, anything can happen

why java c++不同？

java check every time

c++因 performance reason, 不需要 check every time

array length: no run time query to determine length

```
int vector [5] = { 1, 2, 3 ,4 ,5 };
float angles [] = {3.2, 4.2, 4.3 };
char name [] = { 'C', '+', '+' };
```

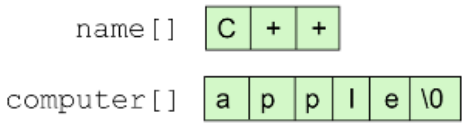
eg:

若*q = vector，q 是 a pointer to int, 指向数组首位(&vector[0])，the value 1

Char Array

“xxx” 这种形式叫 string constants, All string constants 自动末尾追加一个 null character(\0).

```
char name[] = { 'C', '+', '+' };
char computer[] = "apple";
```



char v[] = “apple”;

while(v[i] != ‘\0’)　　不好,因每次找 v[i] 要重新计算 v+i*sizeof(char)，找到该位置的 value

v[2]　　v + 2 * sizeof(char)

v[3]　　v + 3 * sizeof(char)

解决用 pointer：while(*v++ != ‘\0’)

Structures

members of structures　　elements of array

```
struct CarData {
    char reg_no[8];
    char model[30];
    int year;
    float mileage;
};
```

(C++ Struct 与 C 的不同)　　And

CarData mine, yours;

- C++中，mine, yours 是 acture car, value type
- Java 则是 reference type

yours = mine;

- C++ assign 做的是 copy bits，memory 层面的
- Java 则是 copy reference

Union

```
union Merge
{
    int i;
    double d;
    char c;
};
```

- all members share the same location, big enough for largest data
- only one member can be stored at any one time

```
Merge x;
x.d = 3.14;
```

cout << x.d << .c;　　会 Undefined Behaviour

Const Type

value nerve change once initialized

must inti

不可 assigned

Volatile Type

a variable may value changed in way outside the control of compiler (by operation environment, system clock or another program). The qualifier stop compiler optimizing code referring to the variable by storing the variable's value in a register (or read-only memory if volatile const) and re-reading it from there. Compiler must generate code to re-read it from memory, where it may be changed.

volatile const int time {23567689};　　//可改 value 的 const

prevents variable optimization as value may change unexpected

constexpr

```
constexpr int compile_time_only{ 100 };
```

- calculate at compile time, type safe computation
- 是 const
- for objects, implies a stronger form of const

适用场景：

纯 const 并不说明 the value is known at compile time. 但当 some values must be known at compile time 情况, 如: size of array, parameters used in instantiation of template class(eg: vector)

typede/using

造别名 alias

typedef unsigned int UI; UI a;

using UI = unsigned int; (more readable)

Operator

arithmetic * / - + %

relational < > == <= >= !=

logial ! && || (ptr!=0 && *ptr='A') ptr 存在才赋值

bitwise ~ (NOT) << >>(填 0) &(AND, check bit) ^(XOR) |(OR, set 某 bit)

conversion/casting static_cast

unary binary ternary

Assignment Operators

= lvalue = expr

lvalue refer to **the address** in memory of a variable

rvalue refer to **the value** stored in a variable, eg literal constants(3, 'a'…)

+= -= *= /= %= &= ^= |= <<= >>=

Comma Operator

a series of expressions, evaluated from left to right, return the result of last expression

```
result = (++task, a = func(), average);
```

```
while (std::cout << i << std::endl, --i);
```

```
while( std::cin >> ch, ch != '.' )
```

Bitwise Operator

```
int status ()
{
    if (accessMask & activeFlag)
        return someVal;
}
```

flag variables may contain status bits which can be examined by using various masks.

Sequence Points

当某个表达式带有 side-effect（比如改变了一个变量的值），那么它的执行顺序直接影响到了程序执行的结果。

Sequence Point 就是这么一个位置，在它之前所有的 side effect 已经发生，在它之后的所有 side effect 仍未开始，而两个 Sequence Point 之间所有的表达式或者代码执行的顺序是未定义的！

Conversion Operators

casting 多种方法：

```
int i {6};
float f {3.14f};

i = (int) f;                    // C style
i = int (f);                    // C++ functional
i = int {f};                    // C++ functional
                                // disallows narrowing
i = static_cast<int> (f);       // C++ casting ops
```

```
const_cast <T> (expression)
// used to remove the const or volatile of expression
// T data type must be the same as the source type

static_cast <T> (expression)
// Converts expression to the type of Target based on expression

reinterpret_cast <T> (expression)
// allows any pointer to be converted into
// any other pointer type

dynamic_cast <T> (expression)
// casts a from one pointer or reference type to another
// performs a runtime validity check
```

const_cast<>

- Remove const through a pointer, 更建议用 volatile

```
const float f {5.5};
const float *fp {&f};
*(const_cast <float *>(fp)) = 10.5;
std::cout << f << std::endl;
```

static_cast

- 是 general purpose cast, apply at compile time. 但不能 cast struct to int, float to pointer,

可以 cast to and from void*

```
int number {4};
float result { static_cast <float> (number) / 3 };
```

reinterpret_cast

- to cast between incompatiable pointer types
- 可 int to pointer, struct to fioat (implement dependent, affect portability)

```
void foo();

void bar()
{
  int (*f)();

  // f = foo; fail(因void和int type不同)
  f = reinterpret_cast<int (*)()> (foo);
}
```

Useful for low-level programming

Produces implementation dependent code

dynamic_cast it appliers at run-time to polymorphic class hierarchies. down casting

Variables Declaration in If

if(bool c = a || b)　　判断 a||b 同时把结果赋值给 c, 且 return the result。 c 的 scope 在 if{}中

if(c = funA() || funB())　　c 提前赋值好，不重复 call funA 了就

switch

exor 必须是 integer

```
switch (expr)
{
    case branch1:
            statement;
            statement;
    case branch2:
            statement;
            statement;
    ...
    default:
            statement;
            statement;
}
```

Range for statement / Iterator

auto i : v

auto &i : v reference is taken so that each i is an alias to elements in v

begin(), end()

Structuring Programs

declaration //tell compiler just func names and arguments types

main body{

    …

    call func(actual parameter list); //存当前 call 处的 the address，去 stack

}

definition (formal argument list) { //compiler 在这 verify argument types numbers

    //func body, allocate space for variables, (local scoped, on stack)

    …

}

Default Parameter Values

- must shows in function declaration (in a header file, not a separately compiled unit)
- apply from right to left

```
void func(int, int = 0);          declaration
func(1)                           //1, 0
func(1, 3)                        //1, 3
```

在有 default value 的函数时，还提供函数 overload，会造成 ambiguities, 不知 call 哪个

Call by value　　cost！！！

call(A) 传入 (B)　　改 B　　A 不变

- copy parameter values to functions, pass back values
- only can change copy one, no effect on original varaibles

Call by pointer

call(&A) 传入 (*B)　　must 用*B 读 / 改 A

- pass the parameter address to function
- to read /write parameter must dereferencing the address

Call by reference

call(A) 传入 (&B)　　A、B 一回事儿，直接用 B 改 A

- pass parameter aliases(reference) to function
- changes are propagated back directly

和 using pointers 一样高效，还简单，call 时不需要&, 传入也不需要*

但是在 function declaration 时得事先说明该函数 call by value 还是 by reference, 单纯的 func(A)可能 call value or call reference，得看 implement

c++内 & operator meaning

```
&p // take the address of p
a & b // logical AND
a && b // relational AND
int& // an integer reference
```

int i;

std::cin >> i 本质是 cin >> &i

它从 input stream 取下 next integer，把它存入 i，>>符 take a reference to the right hand operand

Inline function

语义上取代 function call site with code(func body) 避免 call overhead(开销), 当 when you call it, compiler 直接用 body of function, with proper arguments and scope 取代 declaration of function

inline 是 c++ 取代 pre-processor #macros 的替代，small function 常用，large function 或有 recursive 的 function，compiler 会忽视 inline 的。

not possible to inline between separately compiled modules, inline functions 常 placed in header files

Macro

macro 传的是 expression，不是 value

```
#include <iostream>
#define MIN(a,b) ((a) < (b) ? (a) : (b))
   (传的是++x, ++y), <号时, cmp 6<7, 返回++x, 即7
int main()
{
    int x=5, y=6;
    std::cout << "min is " << MIN (x,y) << std::endl;
    std::cout << "min is " << MIN (++x, ++y);
}
```

```
min is 5
min is 7
```

```
#include <iostream>
inline int MIN(int a, int b) { return a < b ? a : b;}
   (传值), cmp 6 < 7, 返回6
int main()
{
    int x=5, y=6;
    std::cout << "min is " << MIN (x,y) << std::endl;
    std::cout << "min is " << MIN (++x, ++y);
}
```

```
min is 5
min is 6
```

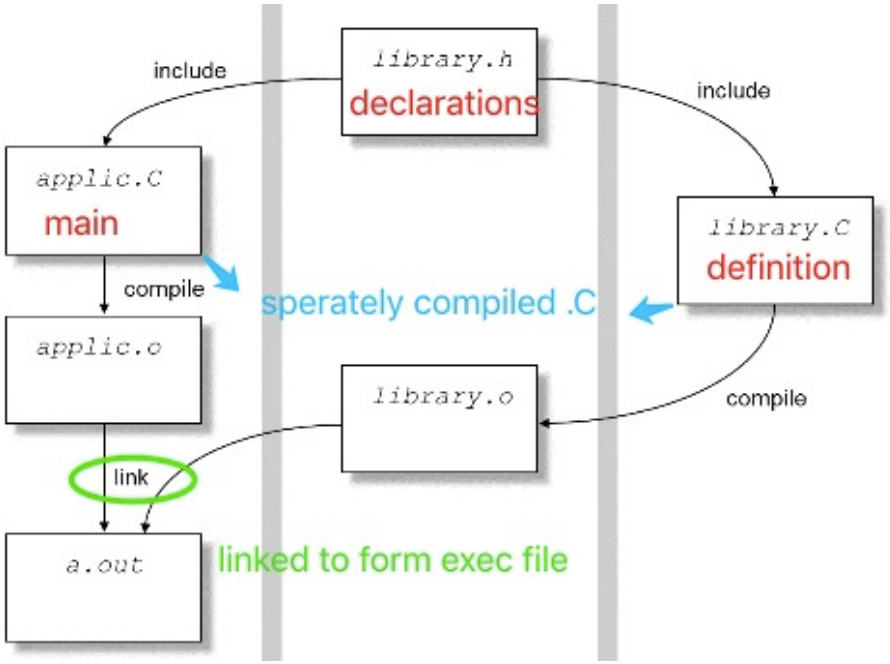
constexpr

function exec at compile time

声明 constexpr 了，for performance reason，compiler 会觉得该函数 not have to appear in generated code 了，just its value when it is used.

g++ -std=c++14 –O3 –S(-S 存汇编语言)

Program Organisation



main.C

```
#include <max.h>

int main()
{
    ...
    total = max (x, y);
}
```

max.h

```
#ifndef MAX_H
#define MAX_H

int max (int, int);

#endif
```

max.C compile

```
#include <max.h>

int max (int a, int b)
{
    return (a > b ? a : b);
}
```

1. main()里#include会先去.h里看declaration
2. compile .C时,生成.Ofile, 含func definition和 external linkage(只linker visible)

1. pre-processor： 通过 include 找所以.h 去保证 content 没有 include multiple times in the same translation unit
2. compile .C 去生成.O object files. it contain func definition with external linkage(visible to linker)
3. 最后 linker link all .o to build executable program.

C++ Linkage

- Mangles function names to support overloading
  - names are augmented with argument types
  - mangled names enable type safe linking

```
void display (int)                    __Z7displayi
float accumulate (float, float*)      __Z10accumulatefPf
```

- To link C code, use C linkage wrapper
  - this stops name mangling so the linker can find the function

```
extern "C"
{
    int atoi (const char *);
    int printf (const char *, ...);
};
```

extern的意思是：别处存在，我这要用

```
extern "C" int atoi (const char *);
```

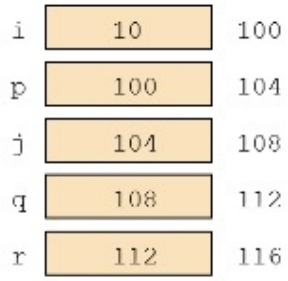
Simple Pointers

Pointers 就是 hold address 的 variables, 与 array 不同

```
#include<iostream>
using namespace std;

int main() {
      int i {5};
      cout << &i << ", "<<i << endl;
      int *p {&i};
      *p = 10;
      cout << &p << ", "<< *p << endl;

      int **j {&p};           //j is a pointer to a pointer to an integer
      int ***q {&j};
      int ****r {&q};
      cout << &j << ", "<< *j<< ", "<< **j << endl;
      cout << &q << ", "<< *q<< ", "<< ***q << endl;
      cout << &r <<  ", "<< *r<< ", "<< ****r << endl;
}                 // stack growing down!!! so, address go down
iaddr = 0x7fff53072a0c, ivalue = 5
paddr = 0x7fff53072a00, *p = i = 10
jaadr = 0x7fff530729f8, *j = iaddr = 0x7fff53072a0c, **j = *iaddr = i = 10
qaddr = 0x7fff530729f0, *q = paddr = 0x7fff53072a00, ** *q = * *paddr =
*iaddr = i = 10
raddr = 0x7fff530729e8, *r = jaddr = 0x7fff530729f8, *** *r = ** *jaddr =
**iaddr = *i = 10
```

```
#include <iostream>

int main()
{
    int i {5};
    int *p {&i};
    *p = 10;
    std::cout << *p << " " << i <<
    std::endl;

    int **j {&p};**值
    int ***q {&j};***值
    int ****r {&q};
}
```

dereference p = i的value



i is an integer

p is a pointer to an integer

j is a pointer to a pointer to an integer

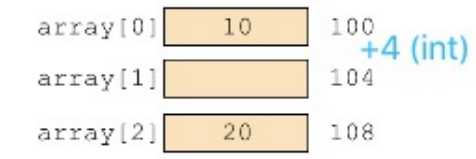
…

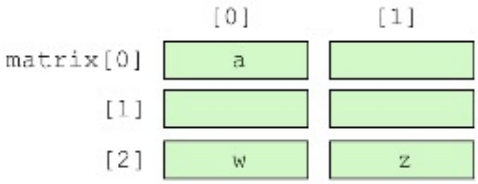
poiner(address) 运算&关系可有：++, --, +, -, =, +=,, -=, ==, <, >, <=, >=, !=, !

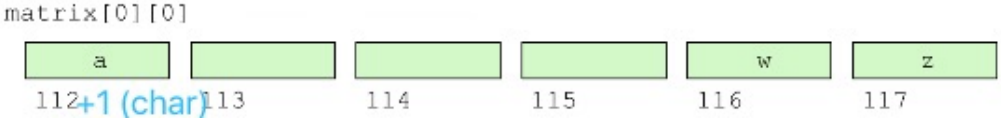
p++，--时，走的单位是 the size of the underlying type. eg integer, 就走 4 bytes

Array

```
#include <iostream>

int main()
{
    int array [3];
    array [0] = 10;
    array [2] = 20;

    char matrix [3][2];
    matrix [0][0] = 'a';
    matrix [2][1] = 'z';
    matrix [2][0] = 'w';
}
```







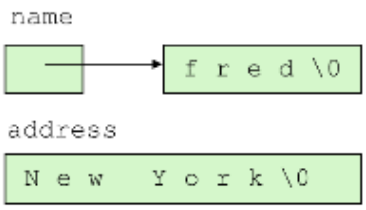
The name of an array is evaluated to the address of the first element.

array == &array[0]

不要混淆 pointer 和 array, 没下标的 array is evaluated to address of first element 所以 confuse 很多人, the array does not contain this information. arrayname 只是个 reference to a location, 不含任何 memory 所以 arrayname 万不可用做 lvalue。

(arrayname += 2)这种是错误的，但 pointer 可行，(pointer += 2)这种是可以的

```
int main()
{
    char *name ("fred");
    char address[] {"New York");
}
```



An array name cannot be used as an l-value because it does not have memory

Pointer Index

When the subscript operator is applied to Pointer p (p[i]), the index given is added as an offset and then the resulting address is dereferenced.

即 p[i] == *(p + i)

Array dereference & index

Array can be dereferenced when the name evaluates to the first element address.

即 *(arrayname + i) == arrayname[i]

When the subscript operator is applied to Array, the index given is added as an offset,

array[i] == &array[0] + i * sizeof(array[i]), 这种运算 performance 不如用指针*p++

Arrays are passed to functions through pointers

call semantics are implicitly call-by-pointer.

In the call, the array is evaluated to the first element address and this is copied to a pointer on the stack of the called function.

func: 以下两种 form 完全等价

void print(char vec[]) == void print(char* vec)

call: print(vector_array);

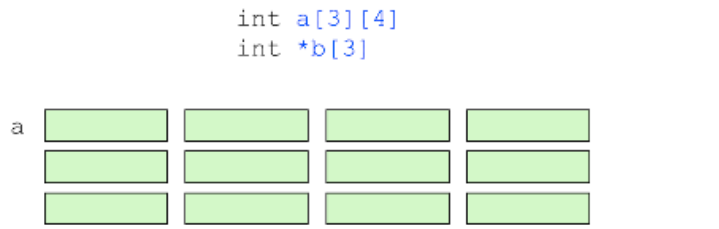
对二维数组的 pass 也一样是穿首元素地址给 pointer on the stack，但同时需要 col 计算 correct offset in matrix.

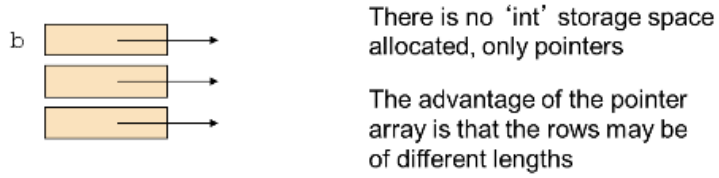
void print(char matrix[][4])

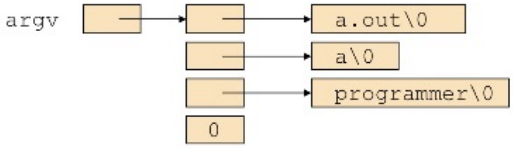
二维以下 2declaration 截然不同：

Array int a[3][4]

PointerArray int *b[3] (row 可 diff length)








```
#include <iostream>

int main (int argc, char *argv[])
{
    std::cout << argc << std::endl;
    while (--argc)
        std::cout << argv[argc] <<
    std::endl;
}
```

argv[0] argv[1] argv[2]

```
$ a.out a programmer
3
programmer
a
```

void pointer

可 point to any type of data

void pointer 可 point to any type　　void *vp{&xxx};

In order to dereferenced void pointer, it must be cast to a real type.

d = *(typeA *) vp　　旧

d = *static_cast<double *> (vp)　　新

cout << (void *) variable;　　print the address of variable

const

- read-only, 必须 init
- Any data type can be const

const char * ptr

char * **const ptr**

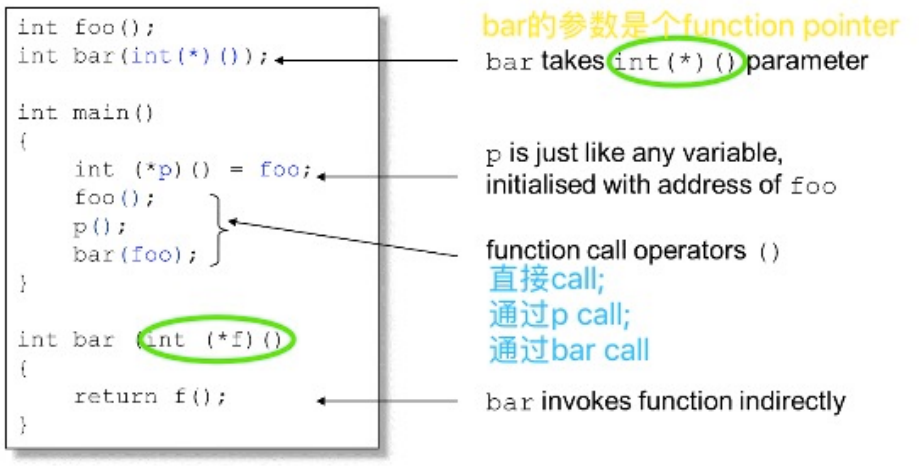
const char * const ptr

const reference 时，如遇到 conversion 情况，an anonymous variable is created to hold the convert value. The reference then refers to the anonymous variable.

Function Pointer

```
int (*f)() = nullptr;
```

- pointer hold the address of a function.
- 可 invoke function 直接从 pointer



Tricky Pointer Definitions

```
#include <iostream>

void foo ();

void (*bar (void (*)()))();

void foo () {
    std::cout << "foo" << std::endl;
}

void (*bar (void (*f)()))() {
    return f;
}

int main () {
    void (*fp)() = foo;
    void (*(*bp)(void (*)()))() = bar;
    bp (fp)();
}
```

```
$ ./a.out
foo
```

```
void (*(*bp)(void (*)()))() = bar
```

use using or typedef to help

```
using FP = void (*)();
typedef FP (*BP)(FP);
FP fp = foo;
BP bp = bar;
bp (fp)();
```

This is a little tricky but is easily resolved when the iterative aspect of the definition is clear. In any case, typedef can be used to simplify.

Note the double function call at bp(fp)()

,
the first returning the address of the function which is then called (using the function call operator) to generate the output foo.

Storage

how data stored in data segment, stack, free store heap

thread_local, extern, mutable

Automatic Objects (func call ~ end)

live in context of a function, eg: function para

created 当入 function， destroyed on leaving

on stack in stack frame data structure(含 return 该返回到哪的 address

Static Object (program begin ~ end)

fixed during exe, exist in data segment

compiler allocates memory for static variables, and ensure all initialized

global V (implicitly static obj) 在 global 可见　　before main alloc, prog starts

static int local (在(scope)里, {}内可见)　　before main alloc, prog starts

init by compiler to 0/init once

reference to static is safe, 因 memory 一直都在

Lookup table, 结合 function pointer build finite state machines

- A function to dispatch a function from a static table
  - useful for building finite state machines

```
#include <iostream>

void zero () { std::cout << "zero" << std::endl; }
void one () { std::cout << "one" << std::endl; }
void two () { std::cout << "two" << std::endl; }
void three () { std::cout << "three" << std::endl; }
void four () { std::cout << "four" << std::endl; }

void foo(int i) {
    static void (*lookup[])() {zero, one, two, three, four};
    lookup[i]();
}

int main() {
    foo (1);
    foo (3);
}
```

function pointer
init with 5 functions的地址

```
$ ./a.out
one
three
```

  - Initialisation of table delayed until first call

Dynamic Memory

heap 由你 custom， new delete

```
int *p {new int};
int *pointer {new int [5]};
```

only data is allocated from the heap. The pointer variable itself is an automatic on the stack frame and hold the address of data.

当 stack goes, pointer goes, 若无 reference 指向 heap 地址，则 fails to read/write/delete, 造成内存 leakage

解决办法：hold the pointer, do what you like, then, delete [] p and set p = nullptr

Object Lifetimes

```
int global;

namespace {
    int file;
};

void func (int arg)
{
    static int stat;

    int local;
    int *p {new int};

    {
        int block;
    }
}
```

	storage class	scope	memory	object lifetime
int global;	static	global	dataseg	program
int file;	static	file	dataseg	program
void func (int arg)	automatic	function	stack	function
static int stat;	static	function	dataseg	program
int local;	automatic	function	stack	function
int *p {new int};	dynamic	pointer	heap	delete
int block;	automatic	block	stack	block

g++ –g *.cpp debug

g++ strip *.cpp deploy

gdb

list

run

stop n

step / step i

set disasm

g++ -q a.out –tui

System Programming

https://ms.semlr.com

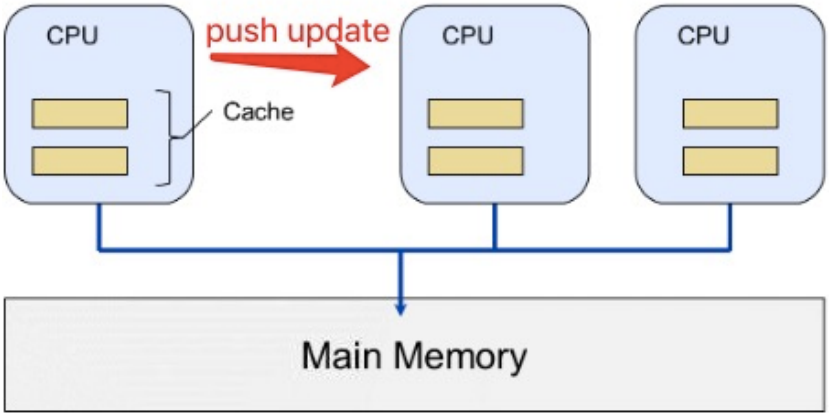
Performance Improving

memory dist 间 data 传输，决定 performance。多年来，人们通过 increase clock rate (speed at which instructions could be executed) 和增大 memory（加 cache） 来提升 performance，但同时造成了 cache misses problems. (where memory needed to be accessed, holding up execution while the data/instruction was fetched.

Share Aemory Multiprocessor Architecture

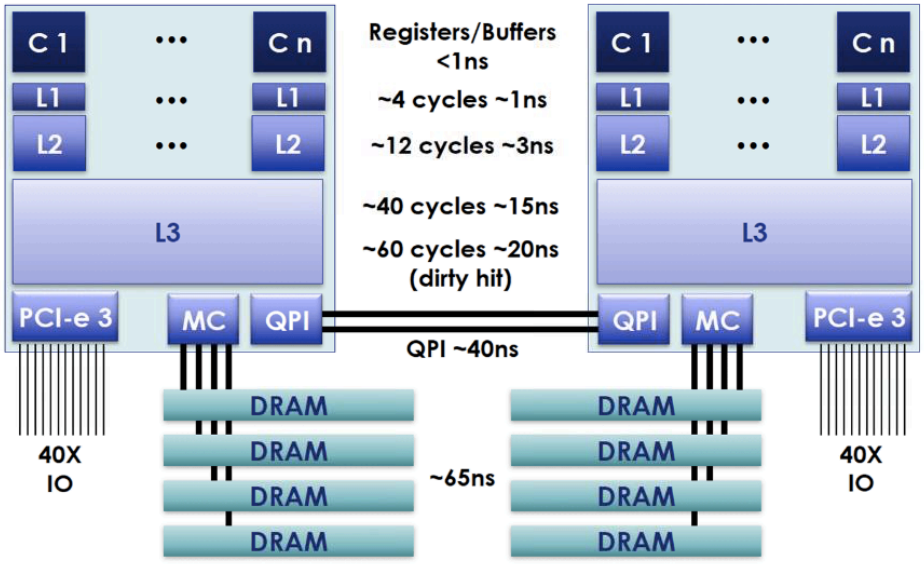
Muiltiple cpus sharing memory

CPU2 要依赖 CPU1 中结果，1 会时刻 push update 给 2 里的 cache。往 main memory 同步有 2 种方式：write through(immediately write to main memory, simple but slow) write back(mark cache line as modified, when line is flushed, write data to main memory, faster and common)



speed for accessing areas of memory

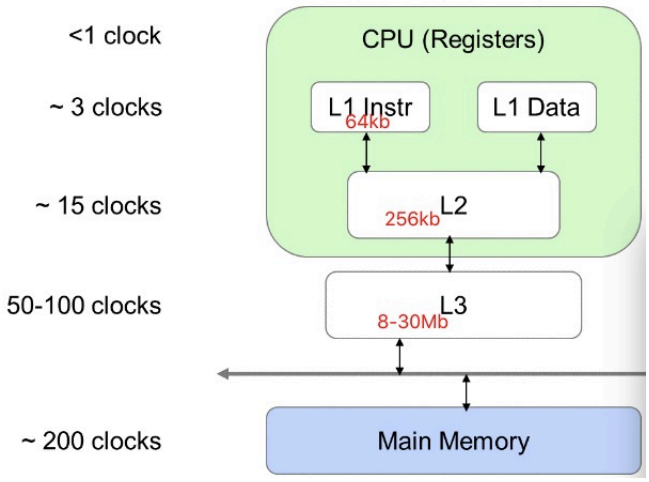
Multicore



- Dynamic RAM
  - usually separate from CPU chip
  - one transistor + one capacitor per bit
  - cheap but slow

- Static RAM
  - often on CPU chip
  - six transistors per bit
  - fast but expensive

Cache 靠近 cpu 越近，space 越小



Cache Coherency/data consistency across caches

MESI protocols: Each line is tagged “Modified Exclusive Shared Invalid”

Memory Barrier: to introduce ordering into memory accesses, load/store

System call

only way user application can invoke functionality within the kernel is by making a system call, eg: read, write, getpid, fork

each system call has a number. When call a system function:

- store system call number
- generate a "trap" so the kernel activates (handler, control to transfer into kernal
- kernel trap handling code recognises “system call” trap
- decode system call number to determine which kernel function to call and what arguments are required
- copy the arguments into the kernel’s memory area
- call the kernel function
- copy any results back to the user process’ memory area
- return the appropriate value to the user process

synchronous I/O: 等 data returned,我才 process

Asynchronous I/O: 边等 data 我边 continue

Tracing a Process

strace 解释 sys calls/signals to processes

```
strace options [-o filename] command
strace options [-o filename] -p pid
```

A Simple example - File Creation

```
#include <iostream>
#include <cstring>
#include <string>
#include <unistd.h>
#include <fcntl.h>

int main() {
        std::string str = "Hello, world";
        int fd = open("Result", O_RDWR | O_CREAT, 0644);
        if ( fd < 0 ) {
                perror("Result");
                exit(1);
        }
        write(fd, str.c_str(), std::strlen(str.c_str()));
        close(fd);
}
```

```
$ g++ Hello.cc
$ a.out
$ strace a.out
… bits of stuff …
open("Result", O_RDWR|O_CREAT, 0644)     = 3
write(3, "Hello, world\n", 13)           = 13
close(3)                                 = 0
…
```

- Summary of times and errors for each call

```
$ strace -c hello
execve("./hello", ["hello"], [/* 64 vars */]) = 0
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
 46.35    0.000146         146         1           write
 22.22    0.000070          23         3           close
 13.33    0.000042          11         4         1 open
  7.30    0.000023           6         4           old_mmap
  2.86    0.000009           9         1           read
  2.54    0.000008           8         1           munmap
  2.22    0.000007           7         1           mprotect
  1.59    0.000005           5         1           uname
  1.27    0.000004           2         2           fstat64
  0.32    0.000001           1         1           brk
------ ----------- ----------- --------- --------- ----------
100.00    0.000315                    19         1 total
```

stace 默认记录 parent process，不记录 child process, 用-f enable child tracing

- Use –f option to enable child tracing
  - use –ff with –o option to put each process' data in separate file

```
$ strace -f a.out
execve("./a.out", ["a.out"], [/* 64 vars */]) = 0
...
fork()                                  = 16623
...
[pid 16622] write(1, "Parent here - the child was 1662"..., 34
          Parent here - the child was 16623 ) = 34
...
write(1, "I am the child\n", 15I am the child
)       = 15
munmap(0x40015000, 4096)                = 0
exit_group(0)                           = ?
```

This part would not appear without -f

If we are saving the output to a file, then we can use the -ff option to help even more. This option forces the output of each process to be placed into separate files, the child (and subsequent children) write into files of the form

filename.pid

where *filename* is the argument supplied for the -o option.

strace -o traceout -ff a.out

would write files

traceout and traceout.pid-of-child

The slide shows the output for the following program:

Timing Information

- Use –r for relative timestamp on call entries
- Use –T to record time spent in calls

```
$ strace -r hello
     0.000000 execve("./hello", ["hello"], [/* 64 vars */]) = 0
     ...
     0.000196 open("Result", O_RDWR|O_CREAT, 0644) = 3
     0.000137 write(3, "Hello, world\n", 13) = 13
     0.000280 close(3)                  = 0
     0.000220 exit_group(0)             = ?
```

```
$ strace -T hello
execve("./hello", ["hello"], [/* 64 vars */]) = 0
uname({sys="Linux", node="annet", ...}) = 0 <0.000010>
...
open("Result", O_RDWR|O_CREAT, 0644)    = 3 <0.000021>
write(3, "Hello, world\n", 13)          = 13 <0.000186>
close(3)                                = 0 <0.000094>
exit_group(0)                           = ?
```

- Sorting the table on any column using the –S flag

```
$ strace -c -S calls hello
execve("./hello", ["hello"], [/* 64 vars */]) = 0
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
 10.80    0.000035           9         4         1 open
  5.56    0.000018           5         4           old_mmap
 25.62    0.000083          28         3           close
  1.23    0.000004           2         2           fstat64
  2.47    0.000008           8         1           read
 48.15    0.000156         156         1           write
  0.31    0.000001           1         1           brk
  2.78    0.000009           9         1           munmap
  0.93    0.000003           3         1           uname
  2.16    0.000007           7         1           mprotect
------ ----------- ----------- --------- --------- ----------
100.00    0.000324                    19         1 total
```

Filtering the Trace

- -e trace=[!]syscall1,syscall2,syscall3
  - only trace the named system calls (! implies negation)
- -e trace=file
  - trace all calls that take a filename as argument
- -e trace=process
  - trace all calls related to process management
- -e trace=network
  - trace all network related calls
- -e trace=ipc
  - trace all IPC related calls

```
$ strace -e trace=open hello
open("/etc/ld.so.preload", O_RDONLY)   = -1
        ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)     = 3
open("/lib/libc.so.6", O_RDONLY)       = 3
open("Result", O_RDWR|O_CREAT, 0644)   = 3
```

Attaching to a Running Process

- Use –p flag

```
$ ls -lR /usr > /dev/null 2>&1 &
[2] 16785
$ strace -e trace=lstat64 -p 16785 -o tout
...
```

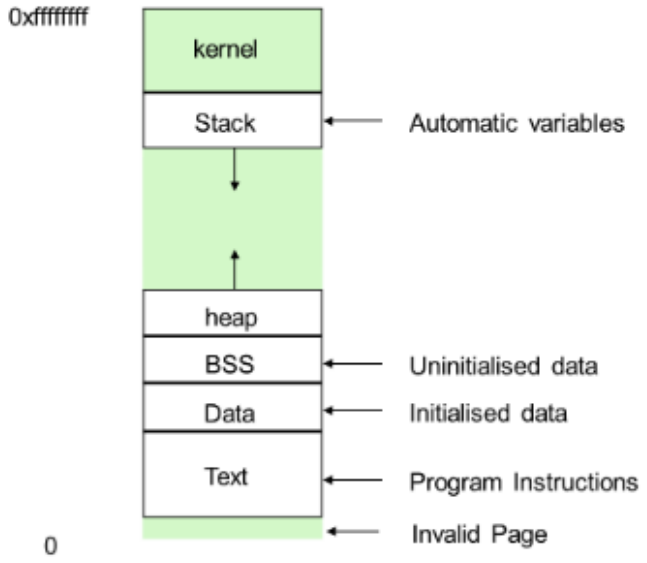
```
$ head -5 tout
lstat64("/usr/lib/perl5/5.8.0/warnings/register.pm",
                {st_mode=S_IFREG|0444, st_size=1014, ...}) = 0
lstat64("/usr/lib/perl5/site_perl/5.6.1",
                {st_mode=S_IFDIR|0755, st_size=536, ...}) = 0
lstat64("/usr/lib/perl5/site_perl/5.8.0",
                {st_mode=S_IFDIR|0755, st_size=720, ...}) = 0
lstat64("/usr/lib/perl5/site_perl/ycp.pm",
                {st_mode=S_IFREG|0644, st_size=37638, ...}) = 0
lstat64("/usr/lib/perl5/site_perl/5.6.1/IO",
                {st_mode=S_IFDIR|0755, st_size=72,...}) = 0
```

Tracing Library Calls – ltrace, It will by default display details of calls to functions in dynamically linked libraries, display arguments and return values.

ltrace hello

ltrace –S hello

Process Memory Layout



File system call

```
#include <unistd.h>
ssize_t read (int fd, void *buffer, size_t count)
  /* returns no of bytes read, 0 on EOF or -1 if an error occurs */

ssize_t write (int fd, void *buffer, size_t count)
  /* returns no of bytes written or -1 if an error occurs */
```

A Simple Copy Program

- Data is read from descriptor 0 into buffer, and then written to descriptor 1.
  - 0 is STDIN and 1 is STDOUT set up before the program runs

```
#include <unistd.h>

#define BUFSIZE 1024

int main ()
{
  char buffer [BUFSIZE];
  ssize_t length;

  while ( (length = read (0, buffer, BUFSIZE)) > 0 )
    write (1, buffer, length);

  return(0);
}
```

```
$ g++ copy.cc
$ a.out
hello
hello
$ cat data
Once upon a time ...
$ a.out < data > other
$ cat other
Once upon a time ...
```

O_RDONLY, O_WRONLY, O_RDWR, O_CREAT

Permissions for newly created file

```
#include <unistd.h>
#include <fcntl.h>

int open (const char *pathname, int flag, mode_t mode);

  /* returns file descriptor if okay or -1 if an error occurs */


int close (int fd);

  /* returns 0 if okay or -1 if an error occurs */
```

Opening and Reading from a File

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFSIZE 1024

int main ()
{
    char buffer [BUFSIZE];
    int length, fd;
    fd = open ("data", O_RDONLY);
    if (fd == -1) {
        perror ("Unable to open data");
        exit(EXIT_FAILURE);
    }
    length = read (fd, buffer, BUFSIZE);
    write (1, buffer, length);
    close (fd);
    return (0);
}
```

```
$ g++ open.cc
$ a.out
Once upon a time ...
```

Creating a File

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

char data[] = "foo bar";

int main () {
  int fd;
  fd = open ("A new file", O_RDWR|O_CREAT, 0644);
  write (fd, data, 8);
  close (fd);
  return (0);
}
```

```
$ g++ creat.cc
$ a.out
$ ls -l
total 33
-rw-r--r--  1 evad        8 Nov  8 19:59 A new file
```

Note the file permissions and file size of 8 bytes

- lseek()
- Allows the 'current file offset' to be modified
  - moves the file pointers anywhere in the file and beyond!
  - forward or backward (negative) offsets allowed

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int direction)

  /* returns new file offset if okay or -1 if an error occurs */
```

- *direction* determines the starting point for lseek

```
SEEK_SET, SEEK_CUR, SEEK_END
```

Process Control

- Four fundamental system calls
  - some variants and related library functions

```
pid_t fork (void)

      returns child PID to parent, 0 to child

int execve (const char *pathname, char *const argv[],
                                  char *const envp[] )

      no return on success, -1 on error

pid_t wait (int *status)

      returns PID of terminated child or -1 on error

void exit (int status)
```

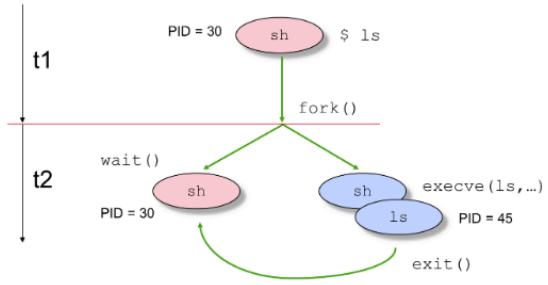
fork creates a new process by replicating the current, calling process. The new process is known as the child, the current process is known as the *parent*. Although called from one process, the parent, it "returns" to both parent and child since they are duplicates and executing the same code.

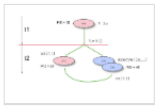
execve loads a new command in the current process. It overwrites the current text, data, bss memory segments with the program specified by *pathname*. No new processes are created. *fork* and *exec* are usually used together.

wait is called by a parent waiting for one of its child processes to terminate. The parent is notified of the childs PID and exit status.

exit should be called by all processes when they terminate to indicate whether the program was successful.

Process Control





The mechanism for creating new processes in Linux is both simple and elegant.

Any process in Linux may create child processes. A particularly obvious example is the shell. The shell starts by reading the command typed in by the user, *ls* in the example above. It then calls the *fork* system call.

fork causes the parent process to duplicate itself. After *fork* there are two processes, identical is every respect except that they have different process identities (PIDs). To the parent process *fork* returns the PID of its child, to the child process *fork* returns zero. Having recognised itself as the parent, the calling process now waits for the child to terminate. It does this using the *wait* system call.

There is no point having two identical copies of the shell. The purpose of *fork* is to copy the environment and attributes of the parent into the child. Upon realising itself to be the child, the child process overlays itself with the program that represents the command the user wishes to execute. This is achieved with the *execve* system call. Overlaying causes the program text of the child shell to be replaced with the text of the new command, but the process is the same.

Once the child has completed its work it returns control to the parent. Provided the parent is waiting for the child to terminate, the child process is now finished. The termination procedure is initiated by the child using the exit system call. This allows the child to pass an exit status back to the waiting parent.

If the child is invoked as a background process the parent does not wait. As a consequence child processes are unable to exit properly because the exit status must be returned to the parent. In such cases the child becomes a zombie.

fork

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main () {
  int pid, fd, status;
  pid = fork ();
  if (pid == 0) { /* Child */
    fd = open("out", O_WRONLY | O_CREAT, 0664);
    status = dup2(fd, 1);
    close(fd);
    execlp("ls", "ls", NULL);
  }
  pid = wait(&status);
}
```

```
$ g++ dup2.cc
$ a.out
$ ls -l
total 33
-rwxrwxr-x  1 evad  24576 Nov  8 19:59 a.out
-rw-rw-r--  1 evad    170 Nov  8 19:24 dup2.c
-rw-rw-r--  1 evad    612 Nov  8 19:39 out
```

This example shows how a program can run a child program with a different standard output; code like this is at the heart of every Linux shell.

The program first calls *fork*. The child process opens a new file, uses *dup2* to ensure that this newly opened file is accessed through descriptor 1 (STDOUT). The child process then uses *execlp* (one of the variants of *execve*) to execute the ls program.

The parent process waits for the child to finish before continuing; a shell would then re-prompt the user for the next command.

exec Functions

- fork creates a new process, but does not run a new program
- exec family of calls load a new program in the process
  - completely replace old program, new program runs from main()

```
int execl ( const char *pathname,
                              const char *arg0, ... )
```

no return on success, -1 on error

```
int execlp ( const char *filename,
                              const char *arg0, ... )
```

no return on success, -1 on error

execlp differs from *execl* in that it is able to follow the PATH environment variable in search of the program file.