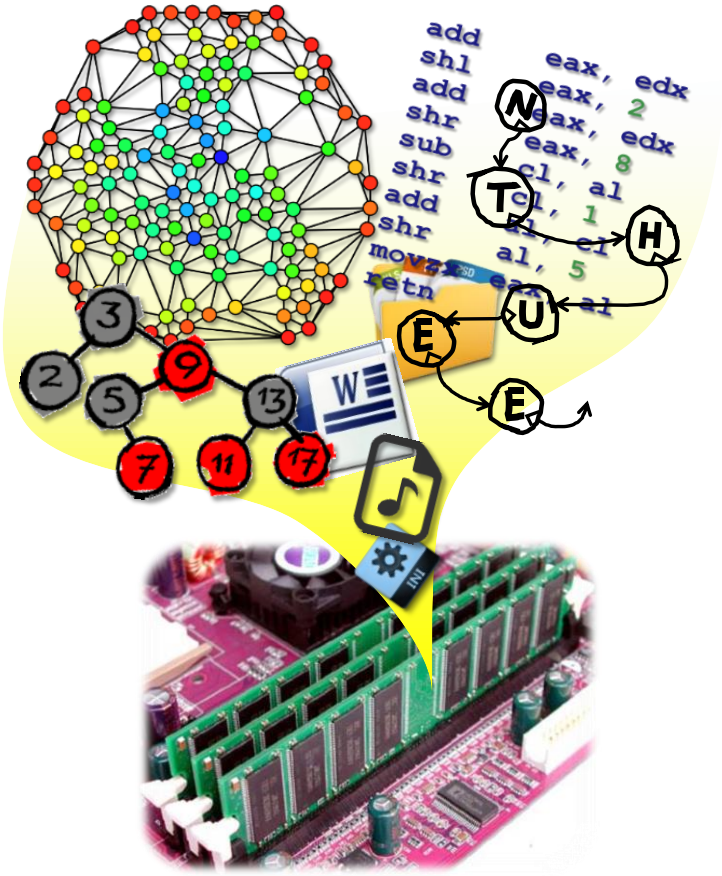
Data Structures

Introduction to C++

Prof. Ren-Shuo Liu

NTHU EE

Spring 2019

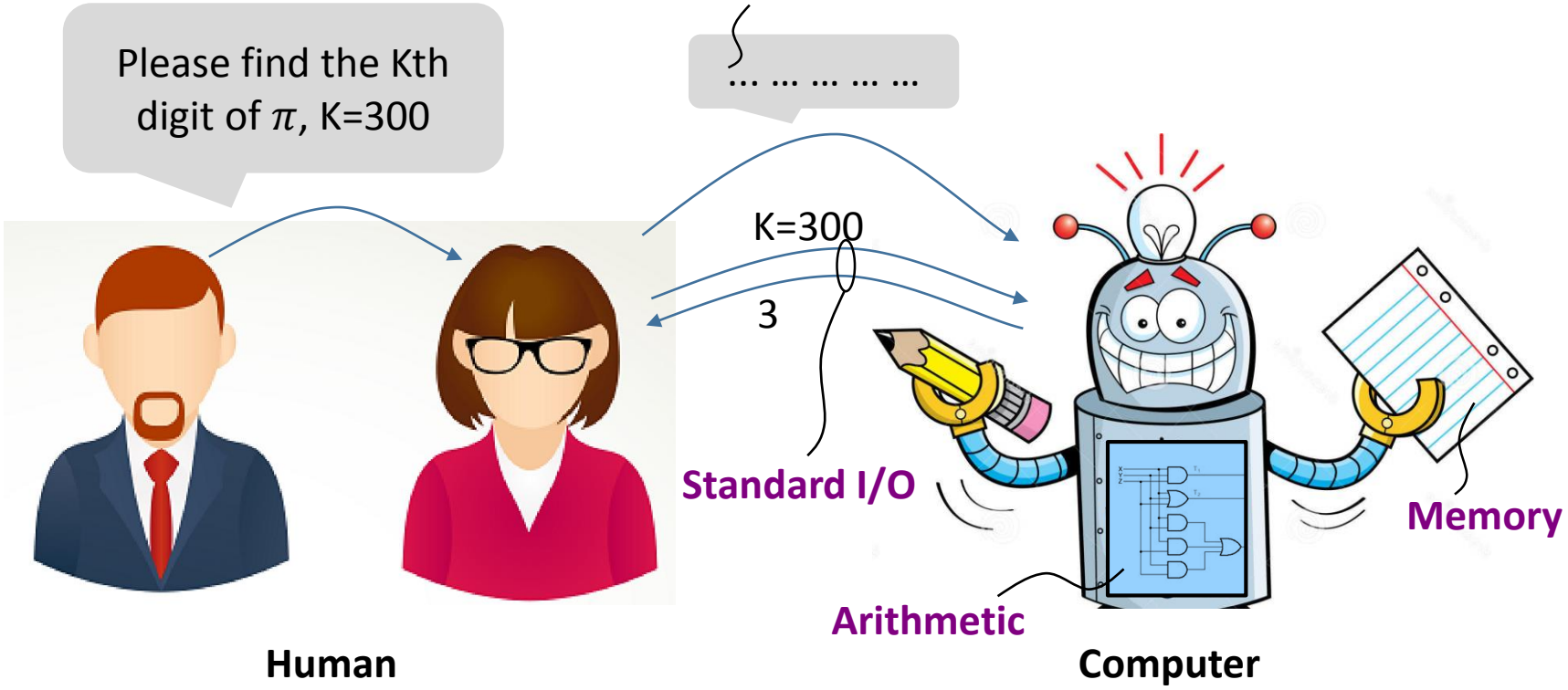


Outline

- **Computer and computer language**
- C++ language
- Variable
- I/O
- Function

Computer and Computer Language

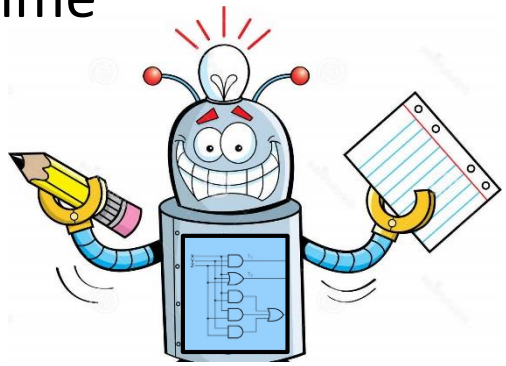
step by step describing how memory, arithmetic, and standard I/O are used to complete the task using **computer language**



Low-Level Computer Language — Machine Code

- Strings of 0s and 1s
  - Typically 8~32 bits per string (i.e., instruction)
  - Instruct computers to perform basic operations
    - Read or write memory
    - Additions, multiplications, etc.
    - Conditional branches
    - …
- Directly understandable by computer
  - Defined at hardware design time
  - Hardware-dependent
- Hardly readable by humans

```
b580
2101
4862
f000 f95b
4862
6841
1c49
6041
bd01
b530
b087
f000 fa8c
```



Low-Level Computer Language — Assembly Language

- Mnemonic (助憶的東西) of machine code
  - A bit better readability for human
  - Non-structured languages
  - Hardware dependent
    - Generally one-to-one correspondence to machine code
- Assembler helps translate assembly language to machine code

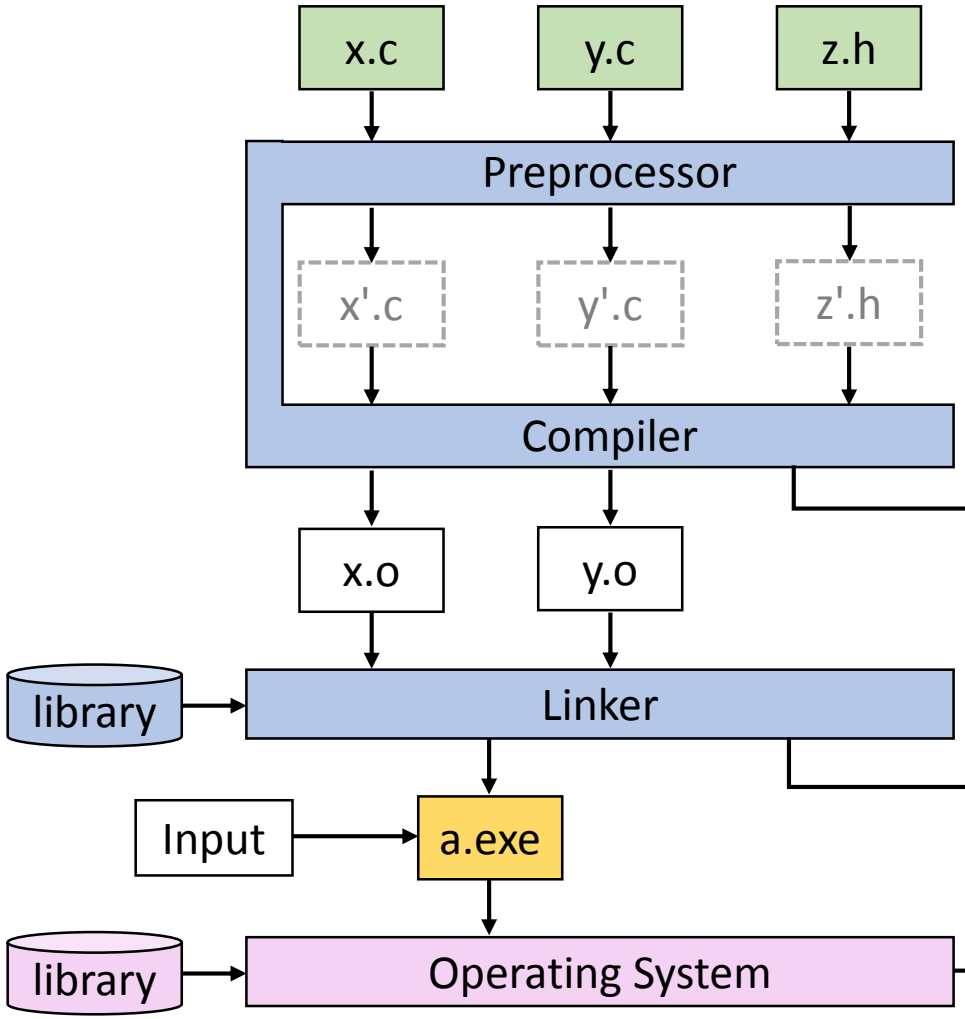
```
PUSH R7, LR
MOVS R1, #1
LDR.N R0
BL 0xf95b
LDR.N R0
LDR R1, [R0, #4]
ADDS R1, R1, #1
STR R1, [R0, #4]
POP R0, PC
PUSH R4, R5, LR
SUB SP, SP, #0x1c
BL 0xfa8c
```

```
b580
2101
4862
f000 f95b
4862
6841
1c49
6041
bd01
b530
b087
f000 fa8c
```

High-Level Computer Language — e.g., C and C++

- Enhanced human understandability
  - English keywords
  - Mathematical notations
- Enhanced programmability
  - Hardware independent
  - Single statement corresponds to multiple low-level instructions

Ecosystems for High Level Language (e.g., C++)



- Compiler errors and linker errors are usually displayed in the same window, but it is important to distinguish these two types of errors

- Compiler warnings
- Compiler errors
- Linker errors

- It is also important to know what runtime errors are

- Runtime errors

Some Exampling Errors

- Compiler errors
  - Something undeclared
  - Jump to case label
  - Multi-line string
- Linker errors
  - Undefined reference to something
- Runtime errors
  - Segmentation fault
  - Overflow / underflow
  - Division by zero
  - Assertion

Something Undeclared

- Example

```
doy.cpp: In function 'int main()':
doy.cpp:25: 'DayOfYear' undeclared (first use this function)
doy.cpp:25: (Each undeclared identifier is reported only once for
each function it appears in.)
doy.cpp:25: parse error before ';' token
```

- Meaning
  - The compiler does not know what "DayOfYear" is
- Common causes
  - Forgot to include the header file that defines the DayOfYear
  - DayOfYear is misspelled
    - at the places where it is defined or used

http://www.csee.umbc.edu/courses/undergraduate/202/fall04/Projects/CommonErrors.shtml

Something Undeclared

- Example

```
xyz.cpp: In function 'int main()':
xyz.cpp:6: 'cout' undeclared (first use this function)
xyz.cpp:6: (Each undeclared identifier is reported only once for
each function it appears in.)
```

- Meaning
  - The compiler does not know what "cout" is
- Common causes
  - Forgot "#include <iostream>"
  - Forgot "using namespace std;"

http://www.csee.umbc.edu/courses/undergraduate/202/fall04/Projects/CommonErrors.shtml

Jump to Case Label

- Example

```
switch.cpp: In function 'int main()':
switch.cpp:14: jump to case label
switch.cpp:11: crosses initialization of 'int y'
```

- Common causes
  - Declared a variable within a "case" inside a switch
- Solution
  - Enclosing code for the case inside of braces

http://www.csee.umbc.edu/courses/undergraduate/202/fall04/Projects/CommonErrors.shtml

Multi-Line String

- Example

Focus on warning or errors that appear first

```
string.cpp:7:12: warning: multi-line string literals are deprecated
string.cpp: In function 'int main()':
string.cpp:7: 'so' undeclared (first use this function)
string.cpp:7: (Each undeclared identifier is reported only once
for each function it appears in.)
string.cpp:7: parse error before 'Mary'
string.cpp:8:28: warning: multi-line string literals are deprecated
string.cpp:8:28: missing terminating " character
string.cpp:7:12: possible start of unterminated string literal
```

- Common causes
  - Missing a quote mark at the end of a string

http://www.csee.umbc.edu/courses/undergraduate/202/fall04/Projects/CommonErrors.shtml

Undefined Reference to Something

- Example

```
/tmp/cc2Q0kRa.o: In function 'main':
/tmp/cc2Q0kRa.o(.text+0x18): undefined reference to 'Print(int)'
collect2: ld returned 1 exit status
```

- Meaning
  - **Linker** cannot find code named Print() that has an int argument in .o files (i.e., compilation is successful)
- Common causes
  - Forgot to declare or define the Print() function
  - Forgot to compile the .c file containing Print()
  - Forgot to link the .o or library containing Print()
  - Misspelled the function name
  - Unmatched argument list

Undefined Reference to Something

- Example

```
/usr/lib/crt1.o: In function '_start':
/usr/lib/crt1.o(.text+0x18): undefined reference to 'main'
collect2: ld returned 1 exit status
```

- Common causes
  - The main function is missing
  - The main function is misspelled

Runtime Error

- Error not shown at the compile/link time
- Usually reported by the operating system
- Usually dependent on input data
  - Some test cases work, but some others do not
- Often dependent on systems
  - Errors can disappear then reappear from time to time

Segmentation Fault

```
int * a = new int [10];
for(int i=0; i<=10; i++) {
  a[i] = i;  // out of the range of a[]
}
…
int sum = 0;
for(int i=0; i<=10; i++) {
  sum += a[i]; // out of the range of a[]
}
cout << "sum = " << sum << endl;
```

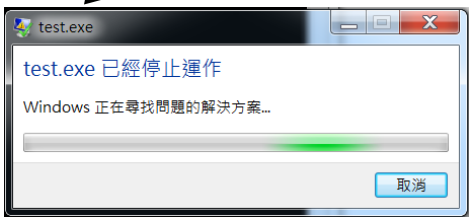
Many outcomes are possible because the contents in memory are corrupted.

```
sum = 55
```

Good result, but it is just a luck. In fact, this is the most dangerous outcome!

```
sum = 90
```

The program executes but a strange result is obtained.
Usually hard to debug.



Runtime error (Segmentation fault)
Usually hard to debug.

Overflow / Underflow


```
int n;
cin >> n;
unsigned int s=0;
for(int i=0; i<n; i++) {
  s += 12345678;
}
cout << "12345678 * " << n << " = " << s << endl;
```

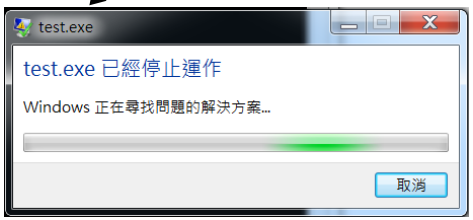
```
1000
12345678 * 1000 = 3755743408
```

Strange result.
No runtime error message.
Usually hard to debug.

Division by Zero

```
int a = 100;
for(int i=0; i<=10; i++) {
  int b;
  cin >> b;
  cout << a/(b-10) << endl; // error occurs once b = 10
}
```

For some inputs, the program works fine

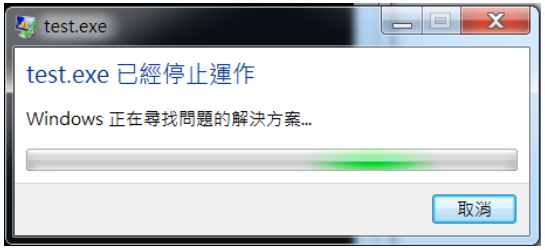


For some other inputs, runtime error occurs (input data dependent)

Assertion

```
#include <assert.h> // assert()
int a = 100;
for(int i=0; i<=10; i++) {
  int b;
  cin >> b;
  assert(b!=10);   // "assert" means "聲明"
  cout << a << "/" << b << " = " << a/(b-10) << endl;
}
```

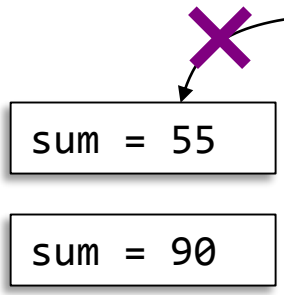
```
15
100/(15-10) = 20
20
100/(20-10) = 10
10
Assertion failed: b!=10, file C:\main.cpp, line 14
```



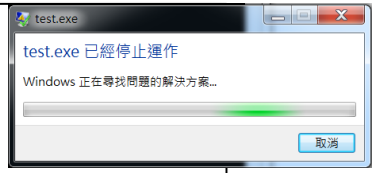
http://www.cplusplus.com/reference/cassert/assert/

Use assert() to Debug Runtime Errors

```
int * a = new int [10];
for(int i=0; i<=10; i++) {
  assert(i>=0 && i<10);
  a[i] = i;  // out of the range of a[]
}
…
int sum = 0;
for(int i=0; i<=10; i++) {
  assert(i>=0 && i<10);
  sum += a[i]; // out of the range of a[]
}
cout << "sum = " << sum << endl;
```




```
Assertion failed: i>=0 && i<10,
file C:\main.cpp, line 12

This application has requested the Runtime
to terminate it in an unusual way.
Please contact the application's support
team for more information.
```



Runtime error

Assertion

- Assertion has no effects if NDEBUG is defined
  - #define NDEBUG
  - Reduce the undesired overhead of assert() for the production version of the program
  - Be careful whether assert() is disabled or not
- It is a good practice to use assert() during program development
  - Assertion helps detecting hard-to-find errors
  - Assertion failure comes with its location information

Assert in Online Judge

- Online Judge shows very limited execution results
  - We cannot distinguish assertion failures from other runtime errors such as division by zero and segmentation faults
  - Thus standard assert() may be not very helpful for debugging
- We can define assert() ourselves

```
#include <stdlib.h>

// trigger "Time Limit Exceed" if assertion fails
#define check_a(x)  if(!(x)) for(;;);

// trigger "Wrong Answer" if assertion fails
#define check_b(x)  if(!(x)) {cout<<"L"<<__LINE__<<" Err" ;exit(0);}
```

C++ Keywords (Vocabulary)

alignas (since C++11)	else	requires (concepts TS)
alignof (since C++11)	enum	return
and	explicit	short
and_eq	export(1)	signed
asm	extern	sizeof
auto(1)	false	static
bitand	float	static_assert (since C++11)
bitor	for	static_cast
bool	friend	struct
break	goto	switch
case	if	template
catch	inline	this
char	int	thread_local (since C++11)
char16_t (since C++11)	long	throw
char32_t (since C++11)	mutable	true
class	namespace	try
compl	new	typedef
concept (concepts TS)	noexcept (since C++11)	typeid
const	not	typename
constexpr (since C++11)	not_eq	union
const_cast	nullptr (since C++11)	unsigned
continue	operator	using(1)
decltype (since C++11)	or	virtual
default(1)	or_eq	void
delete(1)	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
	reinterpret_cast	xor_eq

A total of 86

http://en.cppreference.com/w/cpp/keyword

C++ Keywords (Vocabulary)

alignas (since C++11)	else	requires (concepts TS)
alignof (since C++11)	enum	return
and	explicit	short
and_eq	export(1)	signed
asm	extern	sizeof
auto(1)	false	static
bitand	float	static_assert (since C++11)
bitor	for	static_cast
bool	friend	struct
break	goto	switch
case	if	template
catch	inline	this
char	int	thread_local (since C++11)
char16_t (since C++11)	long	throw
char32_t (since C++11)	mutable	true
class	namespace	try
compl	new	typedef
concept (concepts TS)	noexcept (since C++11)	typeid
const	not	typename
constexpr (since C++11)	not_eq	union
const_cast	nullptr (since C++11)	unsigned
continue	operator	using(1)
decltype (since C++11)	or	virtual
default(1)	or_eq	void
delete(1)	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
	reinterpret_cast	xor_eq

http://en.cppreference.com/w/cpp/keyword

Operators

- Some keyboards lack of special characters, such as &, |, !, ^, ~, thus C++ offers these alternatives

Type	Keywords	Meaning
Logical operators	and	&&
	or	||
	not	!
Bitwise operators	bitand	&
	bitor	|
	xor	^
	compl	~
Compound assignment	and_eq	&=
	or_eq	|=
	xor_eq	^=
Relational and comparison operators	not_eq	!=

Branch and Loop

Type	Keywords
Boolean constants	true
	false
Two-way branch	if
	else
Multi-way branch	switch
	case
	default
	break
Loop	for
	do
	while
Unconditional branch	break
	continue
	return
	goto

```
int s = 0, t = 0;
int n;
cin >> n;
for (;;) {// infinite loop
      // short-circuit evaluation
  if (n > 0 && 100/n)
    s += n;
  else if (n< 0)
    continue;
  else {
    t += n;
    if(t >= 200)
      break;
  }
  cin >> n;
}
cout << s << " " << t;
```

Branch and Loop

Type	Keywords
Boolean constants	true
	false
Two-way branch	if
	else
Multi-way branch	switch
	case
	default
	break
Loop	for
	do
	while
Unconditional branch	break
	continue
	return
	goto

```
int a = 0;
int b = 0;
int upperab = 0;
int others = 0;
char c;
cin >> c;
switch ( c ) {
  case 'A':
    upperab ++;
  case 'a':          //fall through
    a ++;
    break;
  case 'B':
    upperab ++;
  case 'b':          //fall through
    b ++;
    break;
  default:         //catch-all case
    others++;
}
```

Fundamental Types

Type	Keywords
Type with an empty set of values	void
Boolean type	bool
Character type	char
	char16_t (C++11)
	char32_t (C++11)
	wchar_t
Integer type	int
Floating point type	float
	double

Type	Keywords
Size modifiers	short
	long
Signedness modifiers	signed
	unsigned
Storage class specifier	static
CV (const and volatile) type qualifier	const
	volatile
	mutable

Integer Types in Common Systems

Signed Types	Minimum Guaranteed Range	Unsigned Types	Minimum Guaranteed Range	Bits
short short int singed short signed short int	$-2^{15}+1 \sim 2^{15}-1$	unsigned short unsigned short int	$0 \sim 2^{16}-1$	≥16
int signed signed int	$-2^{31}+1 \sim 2^{31}-1$	unsigned unsigned int	$0 \sim 2^{32}-1$	≥32
long long int signed long signed long int	$-2^{31}+1 \sim 2^{31}-1$	unsigned long unsigned long int	$0 \sim 2^{32}-1$	≥32
long long long long int signed long long signed long long int	$-2^{63}+1 \sim 2^{63}-1$	unsigned long long unsigned long long int	$0 \sim 2^{64}-1$	≥64

Fixed Width Integer Types

- #include <cstdint>

	Type name	Min value	Max value
Exact size	int8_t	INT8_MIN	INT8_MAX
	int16_t	INT16_MIN	INT16_MAX
	int32_t	INT32_MIN	INT32_MAX
	int64_t	INT64_MIN	INT64_MAX
Smallest type with at least the required size	int_least8_t	INT_LEAST8_MIN	INT_LEAST8_MAX
	int_least16_t	INT_LEAST16_MIN	INT_LEAST16_MAX
	int_least32_t	INT_LEAST32_MIN	INT_LEAST32_MAX
	int_least64_t	INT_LEAST64_MIN	INT_LEAST64_MAX
Fastest type with at least the required size	int_fast8_t	INT_FAST8_MIN	INT_FAST8_MAX
	int_fast16_t	INT_FAST16_MIN	INT_FAST16_MAX
	int_fast32_t	INT_FAST32_MIN	INT_FAST32_MAX
	int_fast64_t	INT_FAST64_MIN	INT_FAST64_MAX

Unsigned Fixed Width Integer Types

- #include <cstdint>

	Type name	Min value	Max value
Exact size	uint8_t	UINT8_MIN	UINT8_MAX
	uint16_t	UINT16_MIN	UINT16_MAX
	uint32_t	UINT32_MIN	UINT32_MAX
	uint64_t	UINT64_MIN	UINT64_MAX
Smallest type with at least the required size	uint_least8_t	UINT_LEAST8_MIN	UINT_LEAST8_MAX
	uint_least16_t	UINT_LEAST16_MIN	UINT_LEAST16_MAX
	uint_least32_t	UINT_LEAST32_MIN	UINT_LEAST32_MAX
	uint_least64_t	UINT_LEAST64_MIN	UINT_LEAST64_MAX
Fastest type with at least the required size	uint_fast8_t	UINT_FAST8_MIN	UINT_FAST8_MAX
	uint_fast16_t	UINT_FAST16_MIN	UINT_FAST16_MAX
	uint_fast32_t	UINT_FAST32_MIN	UINT_FAST32_MAX
	uint_fast64_t	UINT_FAST64_MIN	UINT_FAST64_MAX

Storage Class Specifier

- Classification based on the memory space holding variable values

Local variable

- **Automatic**: storage is atomically created and discarded per execution of a code segment
- **Static**

Global variable

- **Restricted internal linkage**: storage is only visible by code withing the current C/C++ file
- **External linkage:** storage resides in another C/C++ file of the program

Storage with the specified initial value (if any) is created one time during the entire program life

Dynamic variable: pointers pointing to run-time allocated storage

Storage Class Specifier

- Anatomy according to the memory space holding variable values

Local variable — Declared within one or more curly bracket pairs
- **Automatic** — Others
- **Static** — Using the keyword "static"

Global variable — Declared outside all curly bracket pairs
- **Restricted internal linkage** — Using the keyword "static" (same keyword, different contexts)
- **External linkage** (to global variables in other C files) — Using the keyword "extern"

Dynamic variable — Using the keywords "new" and "delete"

```
int num = 0;  // global variable
static int N; // global variable, restricted accessibility in this C file
int main ()
{
  cin >> N;
  // float y[N];  // wrong
  float * y = new float [N];
  ...
  cout << BinarySearch(y, 3.14, 0, N-1) << endl;
  cout << "Function invocation times: " << num << endl;
  delete[] y;
  return 0;
}
```

main.c

```
int BinarySearch(float * a, const int x, const int left, const int right)
{
  extern int num;  // refer to a global variable outside this C file
  num ++;
  static int num2 = 0;
  num2 ++
  cout << num2 << endl;
  if(left>right) return -1;
  int middle = (left+right)/2;  // automatic local variable
  if(x<a[middle])          return BinarySearch(a, x, left, middle-1);
  else if(x>a[middle])     return BinarySearch(a, x, middle+1, right);
  else                     return middle;
}
```

BinarySearch.c

Variable's Scope

- Scope
  - Visibility of a variable declared in one place and used in another place
  - Each variable has its scope
- Four scope types
  1. File scope
  2. Namespace scope
  3. Local scope
  4. Class scope

```
#include <iostream>
using namespace std;
void func1();
int i = 11;
int j = 22;
int main()
{
   int i = 33;
   int j = 44;
   for(int i=0; i<3; i++){
      int j = i*i;
      cout << i << ", " << j << endl;
   }
   cout << i << ", " << j << endl;
   cout << ::i << ", " << ::j << endl;
   func1();
   return 0;
}
void func1()
{
   int i = 55;
   int j = 66;
   cout << i << ", " <<j << endl;;
}
```

File scope (global variables)

Local scope (in the main function)

Local scope (in the for loop)

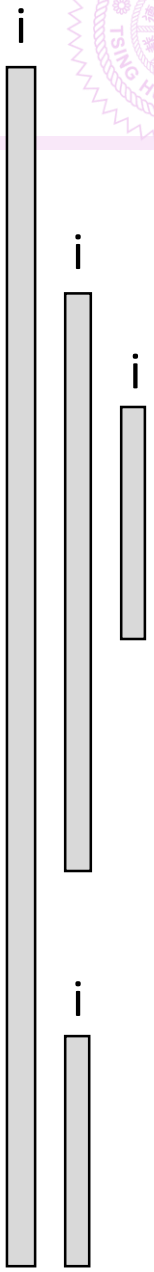
The i in the for loop is used

The ones in the main function are used

The ones of the file scope are referred (because of the :: operator)

Local scope (in the func1 function)

The ones in the func1 are used



```
#include <iostream>
using namespace std;
void func1();
int i = 11;
int j = 22;
int main()
{
   int i = 33;
   int j = 44;
   for(int i=0; i<3; i++){
      int j = i*i;
      cout << i << ", " << j << endl;
   }
   cout << i << ", " << j << endl;
   cout << ::i << ", " << ::j << endl;
   func1();
   return 0;
}
void func1()
{
   int i = 55;
   int j = 66;
   cout << i << ", " <<j << endl;;
}
```

File scope (global variables)

Local scope (in the main function)

Local scope (in the for loop)

The i in the for loop is used

The ones in the main function are used

The ones of the file scope are referred (because of the :: operator)

Local scope (in the func1 function)

The ones in the func1 are used

Output

```
0, 0
1, 1
2, 4
33, 44
11, 22
55, 66
```

Redefinition of Variables


```
void func1();
```

s1.h

```
#include <iostream>
using namespace std;
float PI = 3.14;
void func1()
{
   cout << "func1" << endl;
   cout << "2PI = " << 2*PI;
   cout << endl;
}
```

s1.cpp

```
#include <iostream>
#include "s1.h"
using namespace std;
float PI = 3.14;
int main()
{
   cout << "main" << endl;
   cout << "PI = " << PI << endl;
   func1();
   return 0;
}
```

main.cpp

Linker error

- A linker is designed to maintain a table collecting the source definition of global variables
- Since the name, PI, is defined from two sources, a linker complains that PI is defined multiple times.

Link to External Variables


```
void func1();
```

s1.h

```
#include <iostream>
#include "s1.h"
using namespace std;
float PI = 3.14;
int main()
{
   cout << "main" << endl;
   cout << "PI = " << PI << endl;
   func1();
   return 0;
}
```

main.cpp

```
main
PI = 3.14
func1
2PI = 6.28
```

Output

```
#include <iostream>
using namespace std;
float PI =-3.14;
extern float PI;
void func1()
{
   cout << "func1" << endl;
   cout << "2PI = " << 2*PI;
   cout << endl;
}
```

s1.cpp

If we mean to refer to the predefined variable external to a source file, we need to use the keyword "extern" to explicitly show this intention.

“Pure” File-Scope Variables


```
void func1();
```

s1.h

```
#include <iostream>
using namespace std;
static float PI = 5;
void func1()
{
   cout << "func1" << endl;
   cout << "2PI = " << 2*PI;
   cout << endl;
}
```

s1.cpp

```
#include <iostream>
#include "s1.h"
using namespace std;
float PI = 3.14;
int main()
{
   cout << "main" << endl;
   cout << "PI = " << PI << endl;
   func1();
   return 0;
}
```

main.cpp

```
main
PI = 3.14
func1
2PI = 10
```

Output

If we mean to defined a variable local to a source file, one solution is to use the keyword "static" to explicitly show this intention.

Resolving Name Collisions


```
void func1();
```

s1.h

```
#include <iostream>
using namespace std;
float s1_PI=5;
void func1()
{
   cout << "func1" << endl;
   cout << "2PI = "
        << 2 * s1_PI;
   cout << endl;
}
```

s1.cpp

```
#include <iostream>
#include "s1.h"
using namespace std;
float PI = 3.14;
int main()
{
   cout << "main" << endl;
   cout << "PI = " << PI << endl;
   func1();
   return 0;
}
```

main.cpp

```
main
PI = 3.14
func1
2PI = 10
```

Output

Another solution is to rename the variable (e.g., adding a prefix "s1" in this example). Doing this of course avoids name collisions.

Resolving Name Collisions


```
void func1();
```

s1.h

```
#include <iostream>
using namespace std;
namespace s1{
float PI=5;
void func1()
{
   cout << "func1" << endl;
   cout << "2PI = " << 2*PI;
   cout << endl;
}
}
```

s1.cpp

```
#include <iostream>
#include "s1.h"
using namespace std;
float PI = 3.14;
int main()
{
   cout << "main" << endl;
   cout << "PI = " << PI << endl;
   s1::func1();
   return 0;
}
```

main.cpp

```
main
PI = 3.14
func1
2PI = 10
```

Output

Another solution is to use the keyword "namespace" to specify a name (e.g.,"s1" in this example. The name needs NOT be the file name). Doing this lets the associated variables and functions have a namespace scope and avoid name collisions.

Value, Address, Pointer, and Reference

- Meanings of '&' and '*' are context dependent
  - This is one main reason why many people have difficulties in understanding pointers and references in C++

- Context is common
  - 目的地是清大
  - 你的確在清大
  - 他的卻在交大

- **Declarations** about variables
  - **int** a
    - Declare that a is an integer
  - **int *** p
    - Declare that p is a pointer. Specifically, a pointer to an integer
  - **int &** r
    - Declare that r is an alias (別名) of an integer

- **Operations** on variables
  - &a
    - Get the address of a
  - *p
    - Get the thing that p points to
      - If p is declared as int *, get the int
      - If p is declared as float *, get the float

- **Declarations** (declare some thing)
  - int a
  - int * p
  - int & r
- **Operations** (get something)
  - &a
  - *p

- **Declarations**
  - int reference
  - int * value
  - int & pointer
- **Operations**
  - &address
  - *data

- float tmp
- float & name
- &address
- *alias
- int pointer

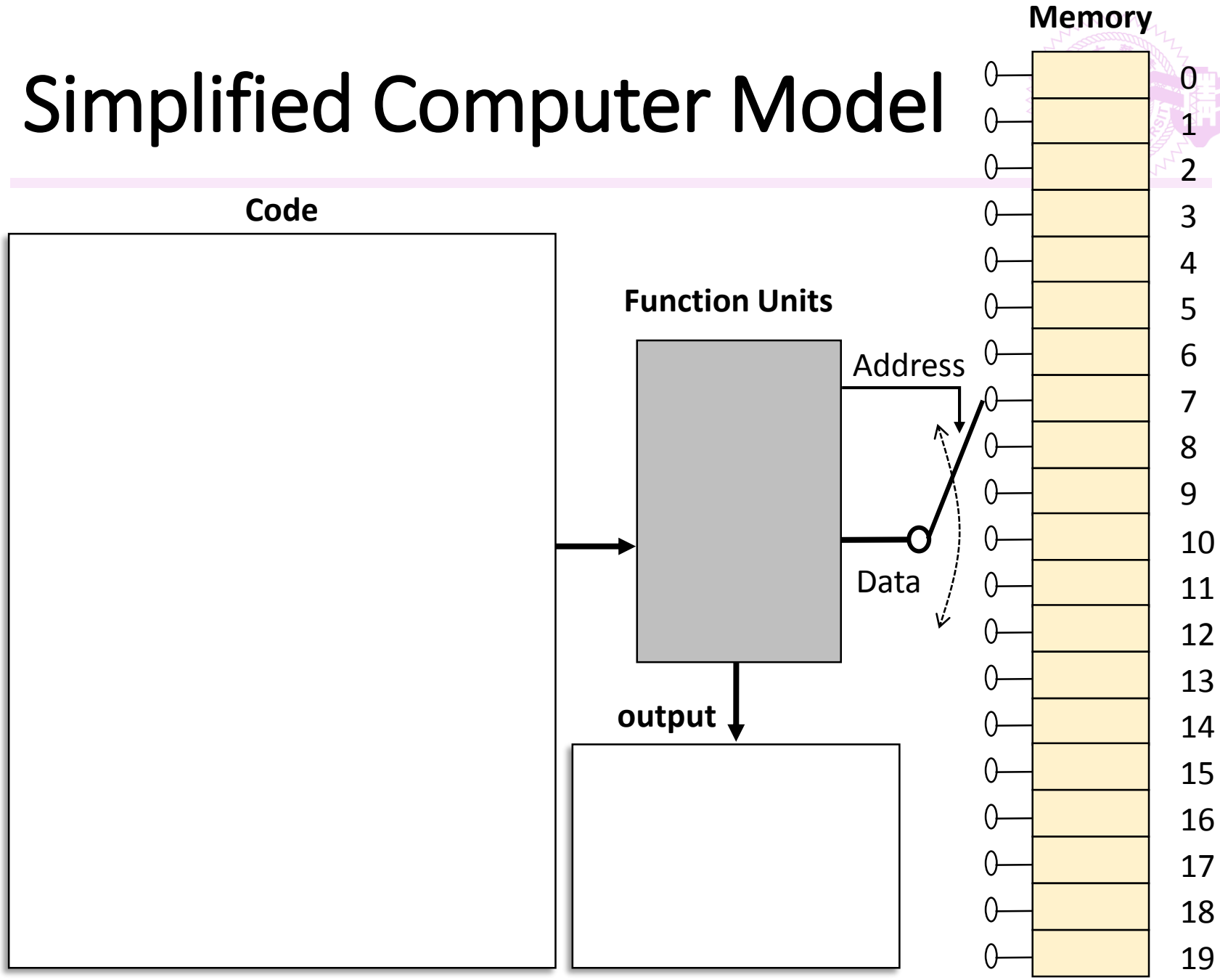
- if (*people == 10)
- if (2 *people == 20)
- float & integer
- if (K & R)
- if (& R)
- int * character

- float alias
- switch(& hello)
- x = y * (*z)
- & both + *none
- char * float_array

- **double type**
- & where
- int **&address**
- char &p_char
- & p_**char**

- *non_address
- int *ref
- *int &* p
- &ref
- int function

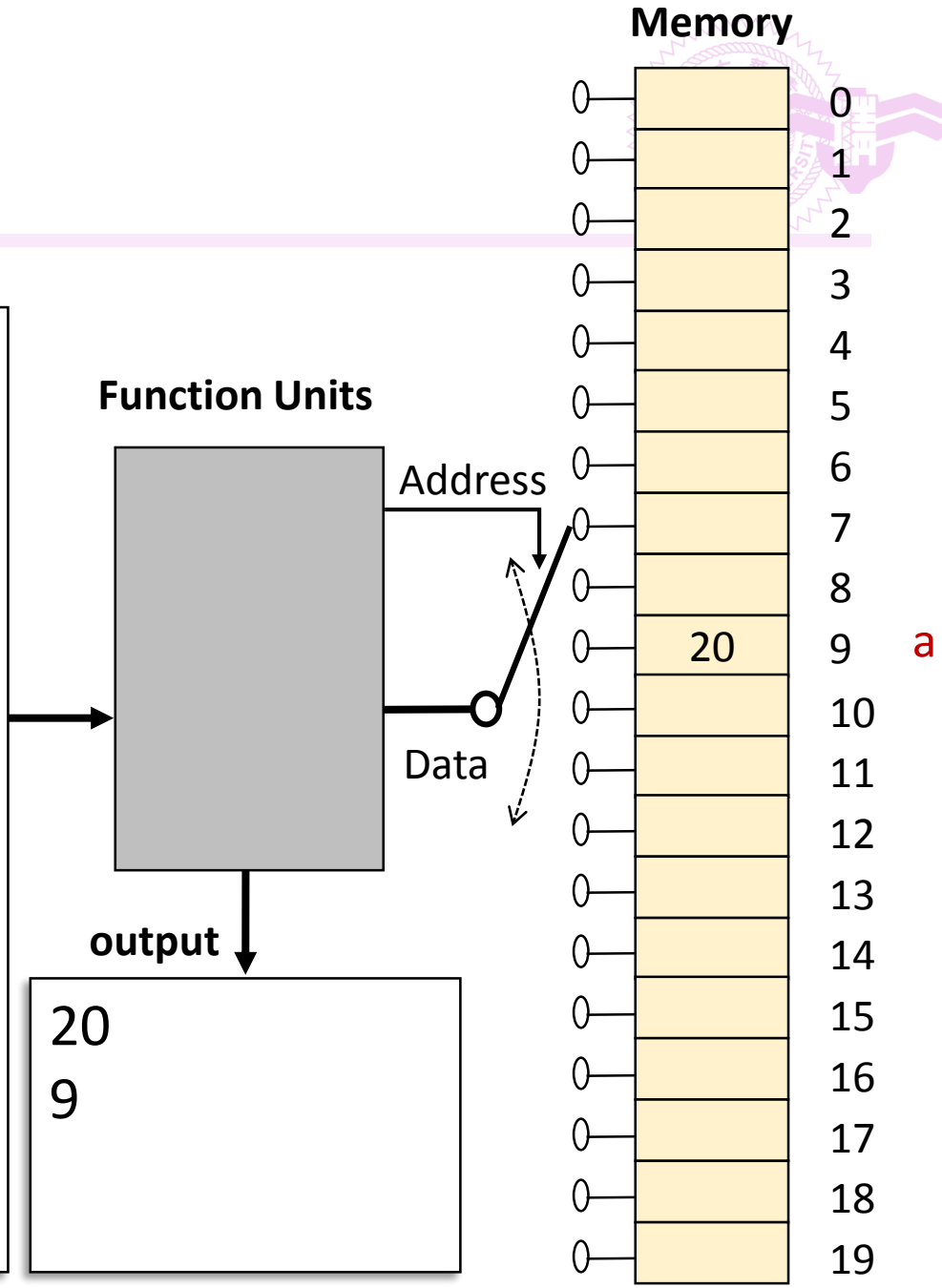
Simplified Computer Model



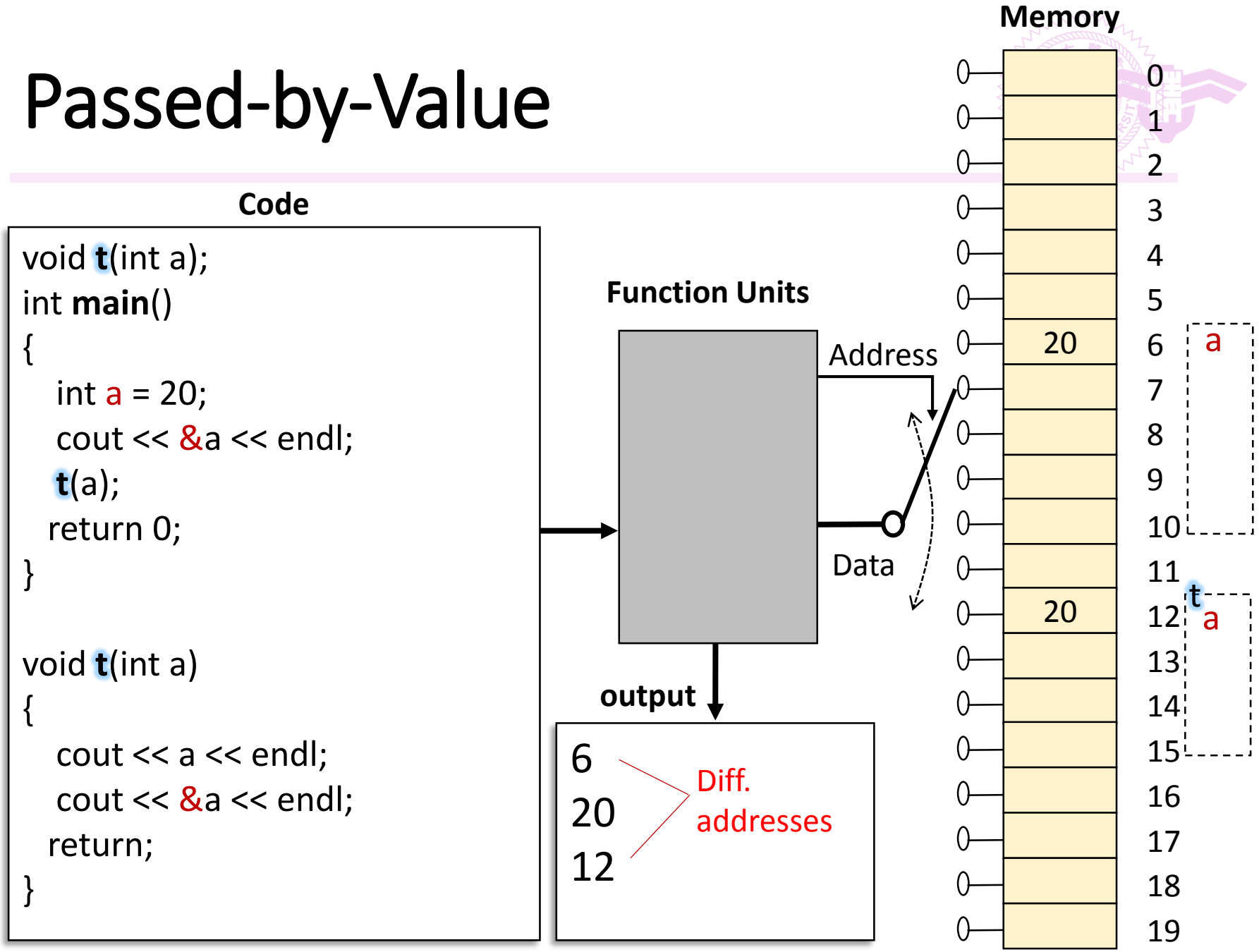
Address

Code

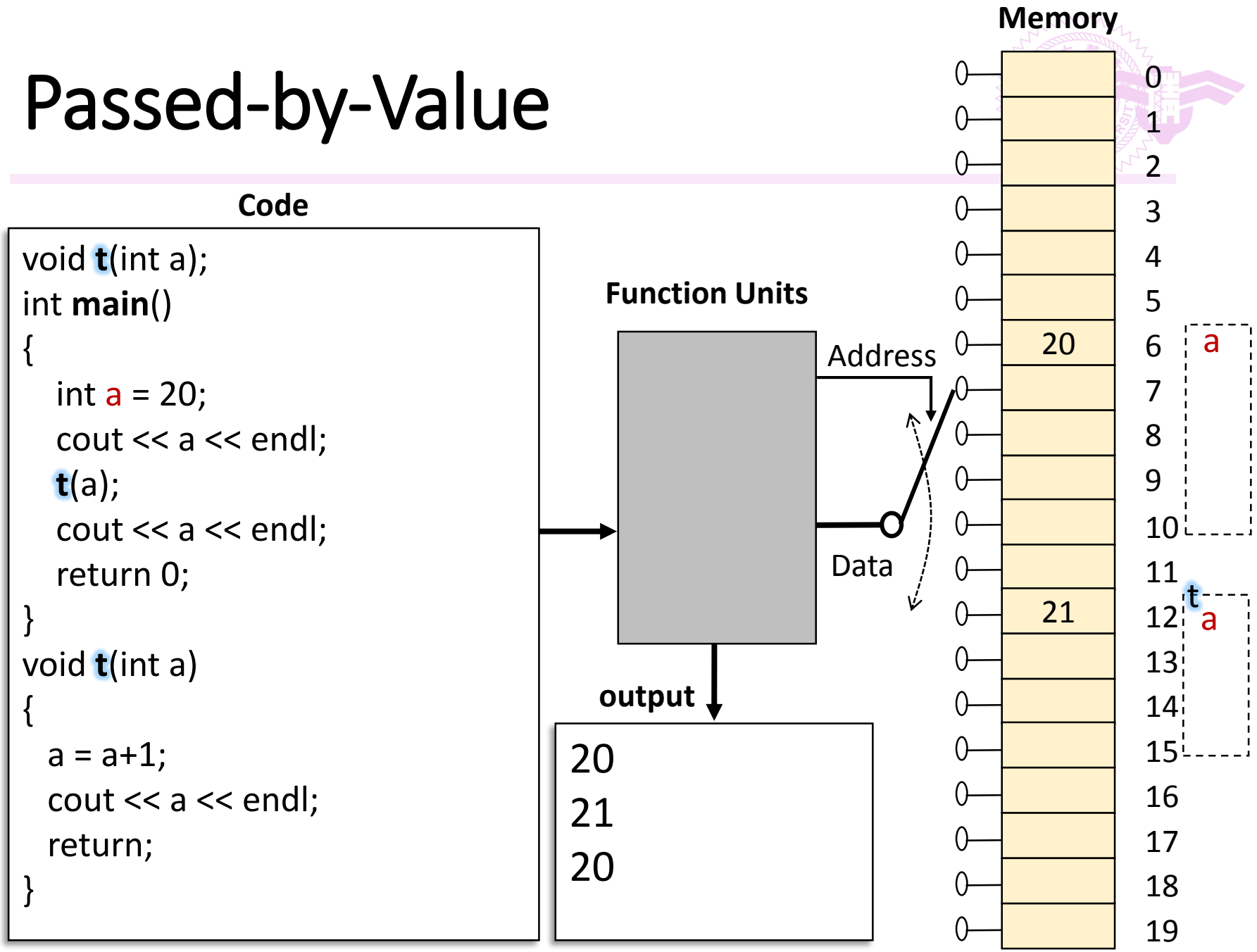
```
int main()
{
    int a;
    a = 20;
    cout << a << endl;
    cout << &a << endl;
    return 0;
}
```



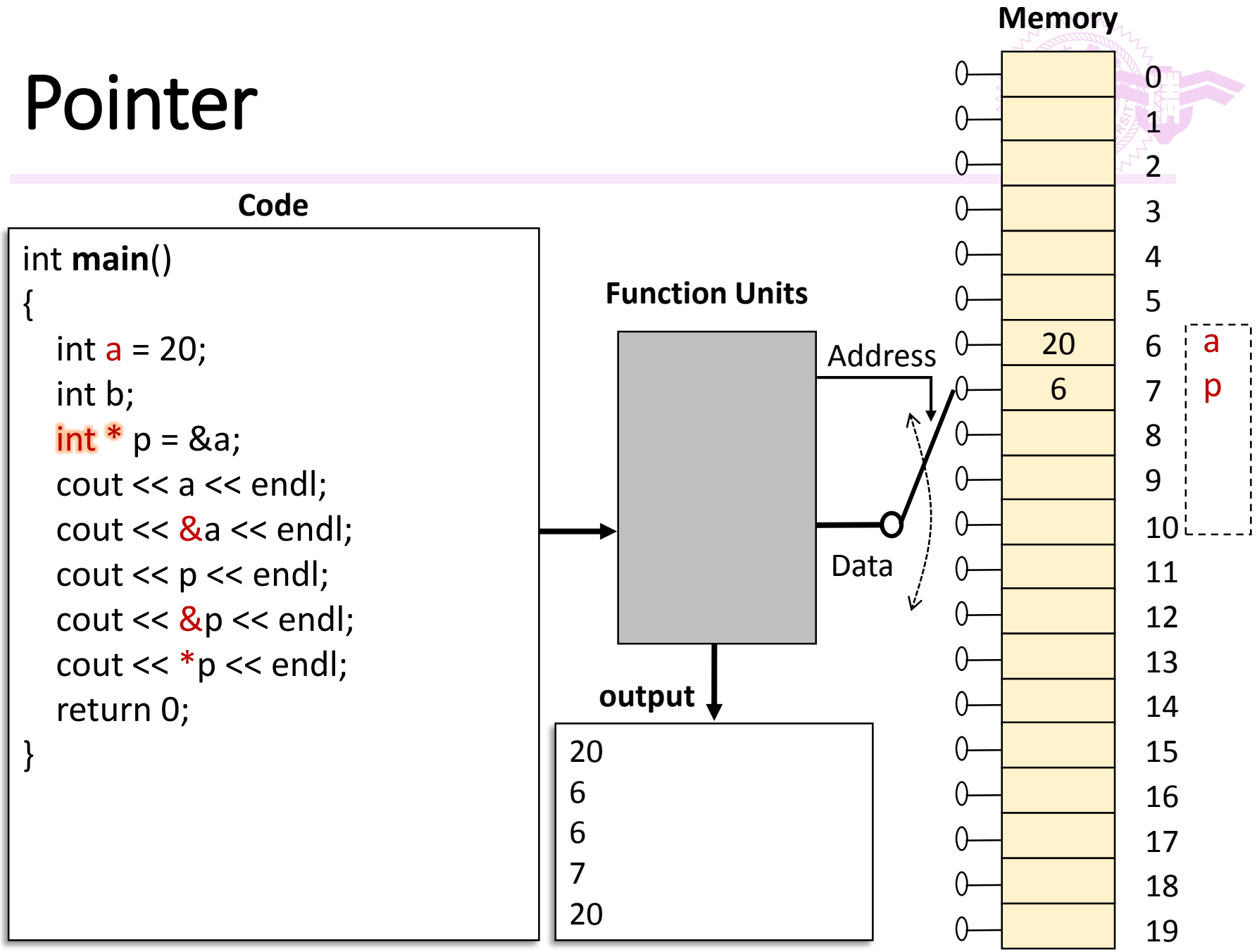
Passed-by-Value



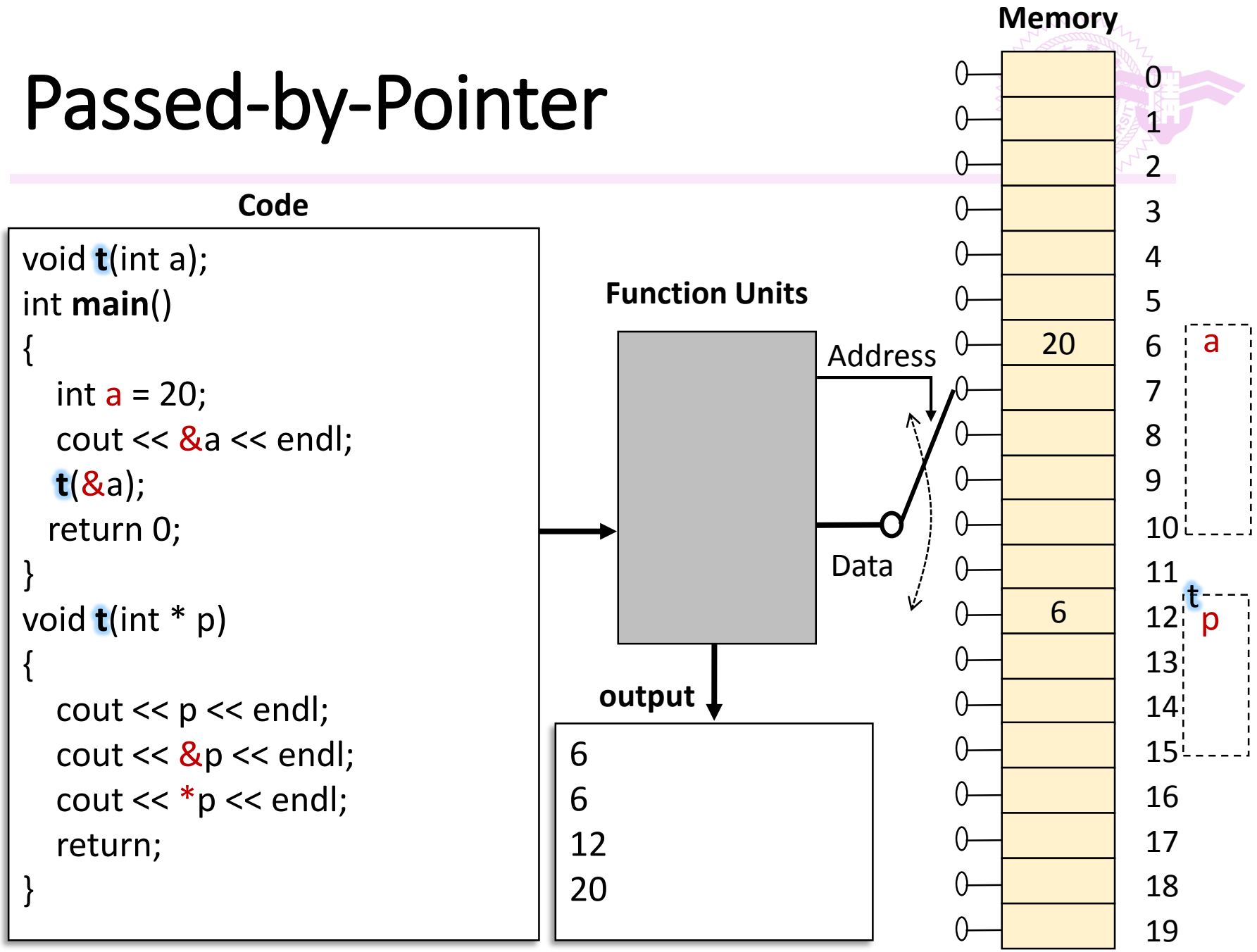
Passed-by-Value



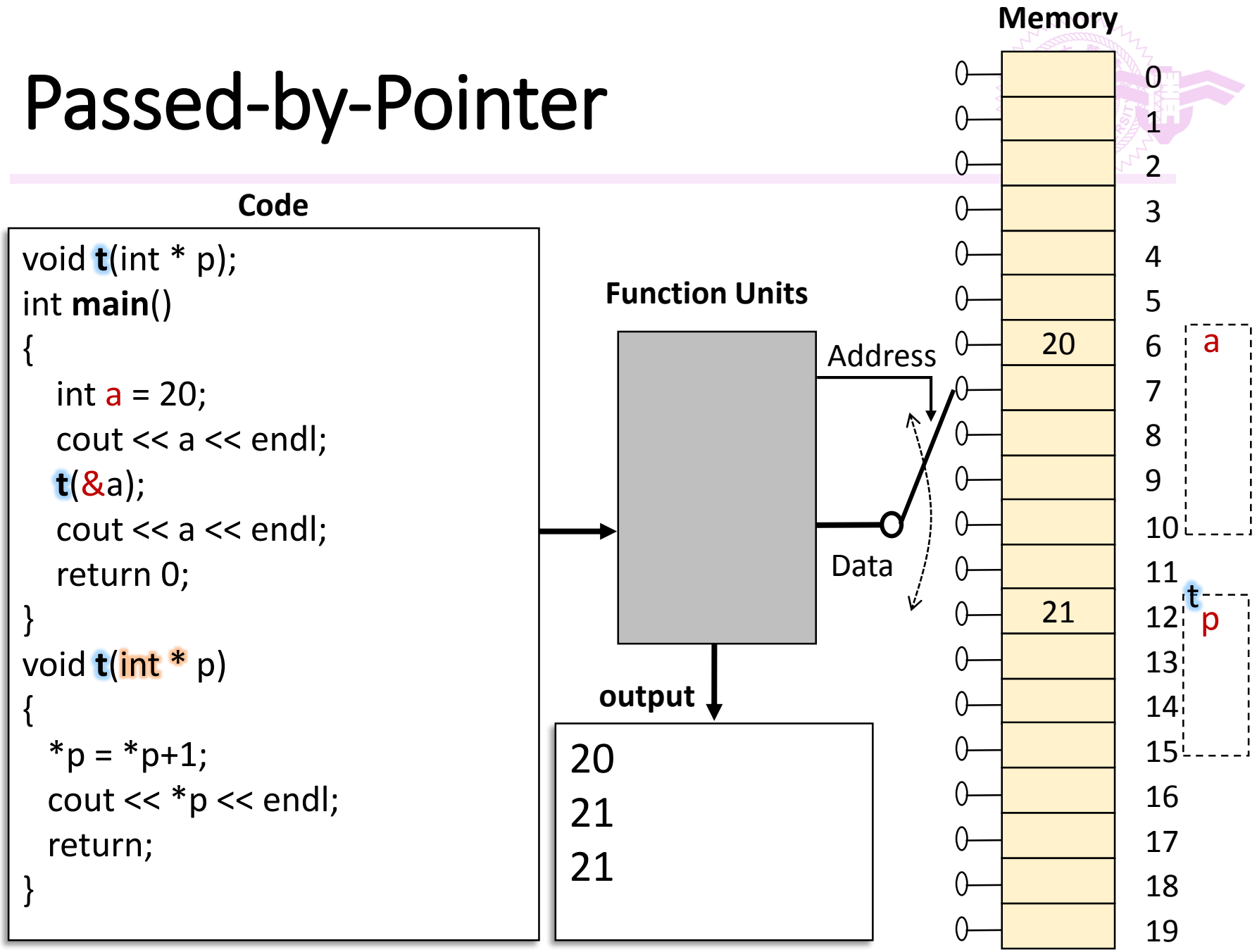
Pointer


Code
int main()
{
int a = 20;
int b;
int * p = &a;
cout << a << endl;
cout << &a << endl;
cout << p << endl;
cout << &p << endl;
cout << *p << endl;
return 0;
}
Function Units
Address
Data
output
20
6
6
7
20
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
6
a
p

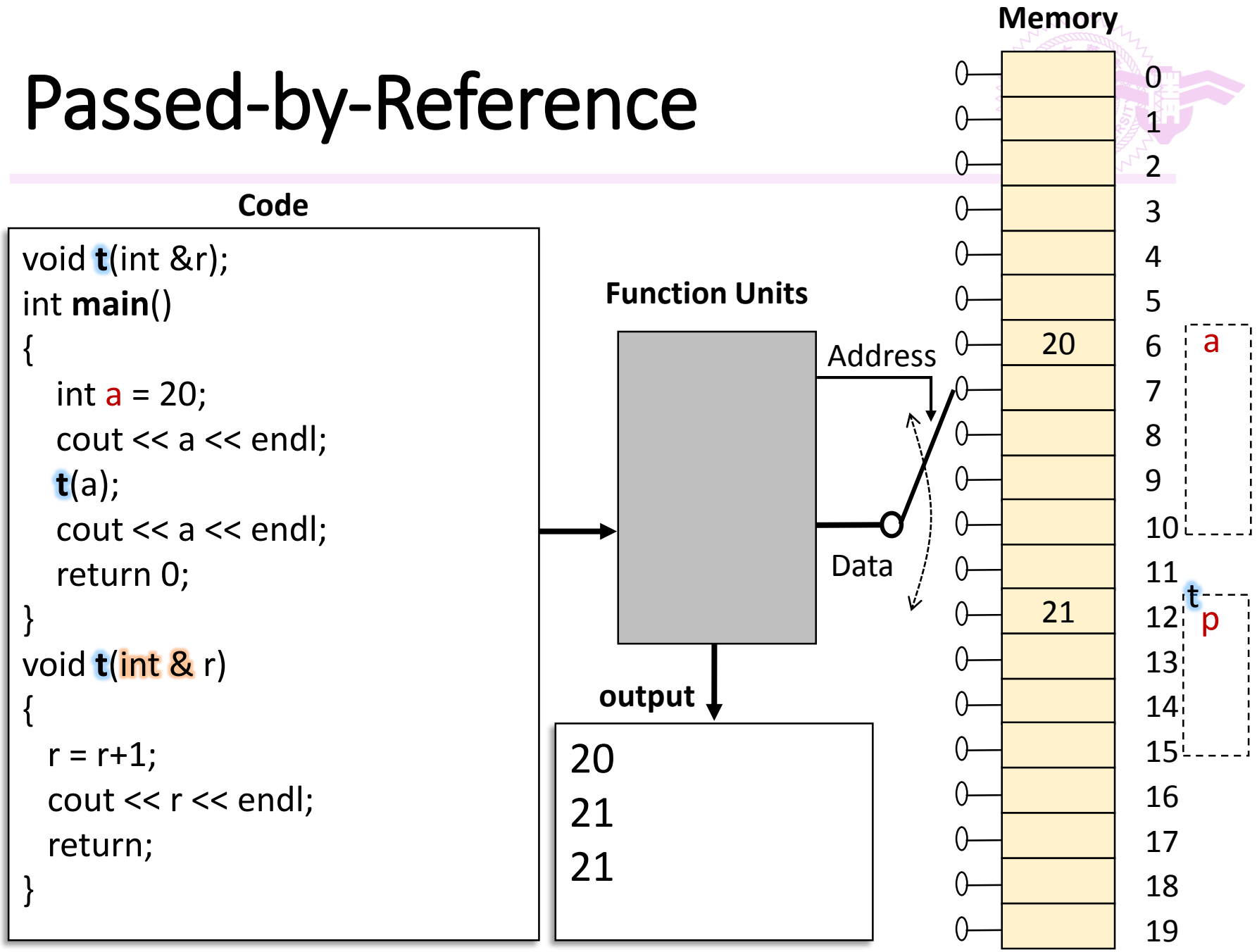
Passed-by-Pointer



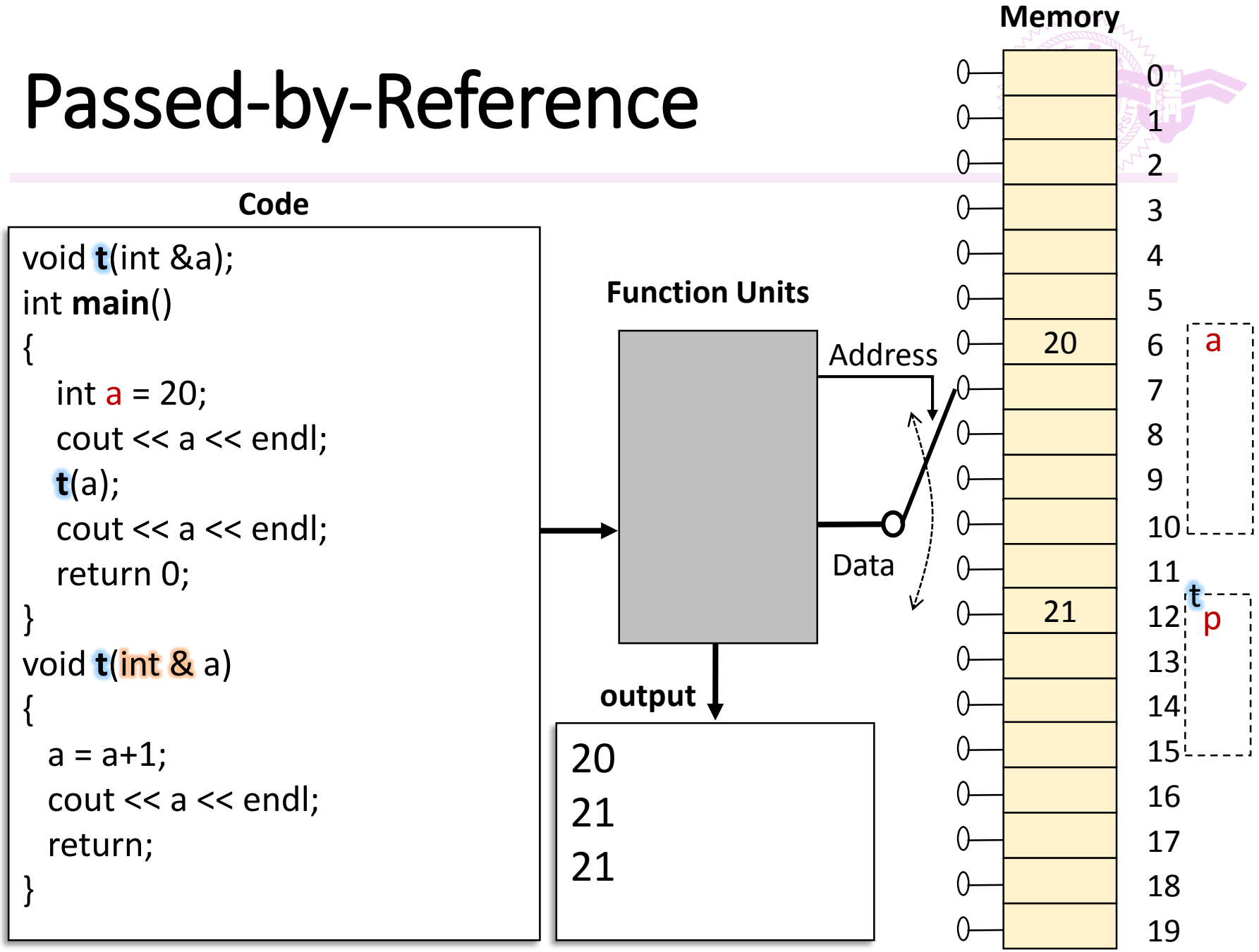
Passed-by-Pointer



Passed-by-Reference



Passed-by-Reference



Comments

- Two approaches
  - Multiline comments
    - /* comments
      * comments
      * comments
      */
    - Nested multiline comments is not supported
  - Single line comments (new features in C++)
    - // comments

```
#include <iostream>
using namespace std;
int main()
{
   int a = 0;
   /*
      /*
       a = 1;
      */
   a = 5;
   */
   cout << a << endl;
   return 0;
}
```

Compile error due to dangling */

```
#include <iostream>
using namespace std;
int main()
{
   int nest = /*/*/0*/**/1;
   cout << nest << endl;
   return 0;
}
```

No error but potentially a bug

Input Output

- Include the system-defined header file *iostream*
- Use namespace *std*
  - Because *cin* and *cout* are in that namespace
- C++ I/O can be format-free

```
#include <iostream>
using namespace std;
int main()
{
   int a;
   float b;
   cout << "please enter a int and a float:"
        << endl;
   cin >> a >> b;
   cout << a << ", " << b << endl;
return 0;
}
```

```
please enter a int and a float:
1 0.5
1, 0.5
```

File Input Output

- Include the system-defined header file *fstream*

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ofstream fout("my.txt");
   if(!fout) {
      // standard error device
      cerr << "cannot open my.txt" << endl;
      return 1;
   }
   int n = 50;
   float f = 20.3;
   fout << n << endl;
   fout << f << endl;

   return 0;
}
```

File Input Output (cont'd)

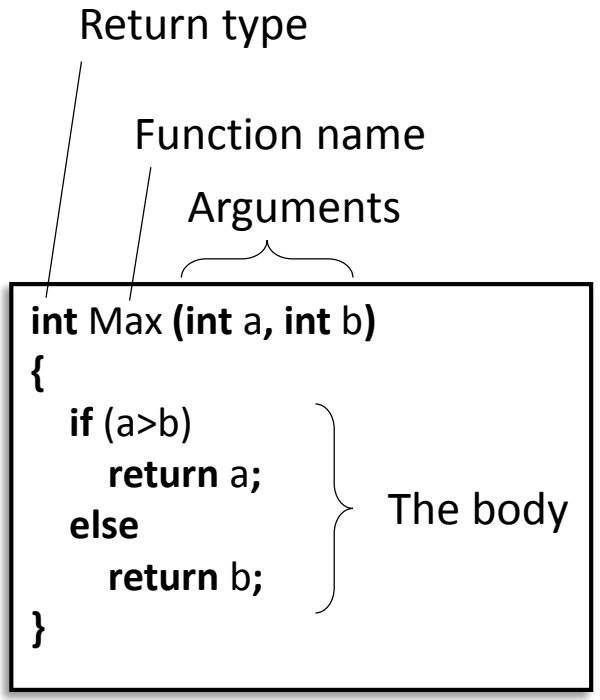

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ifstream fin("my.txt");
   if(!fin) {
      // standard error device
      cerr << "cannot open my.txt" << endl;
      return 1;
   }
   int n;
   float f;
   fin >> n >> f;

   cout << "n is " << n << endl;
   cout << "f is " << f << endl;
   return 0;
}
```

```
n is 50
f is 20.3
```

Screen Output

Functions

- Two kinds
  - Regular functions
  - Member functions
    - Associated with specific C++ classes
- Function consists of
  - A function name
  - A list of arguments (aka signature)
    - We shall see why this list is called signature shortly
  - A return type
  - The body
    - Code that implements function



Functions

- Return types
  - If a function is not meant to return anything
    - Use *void* to denote the return type
  - In C++, *main* function must return an *int*
- Common practices use *main* function's return value to indicate how a program exited
  - Zero means normal exit
  - Non-zero values are error codes that indicate abnormal termination

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ifstream fin("my.txt");
   if(!fin) {
      // standard error device
      cerr << "cannot open my.txt" << endl;
      return 1;
   }
   int n;
   float f;
   fin >> n >> f;

   cout << "n is " << n << endl;
   cout << "f is " << f << endl;
   return 0;
}
```

Function Name Overloading

- C++ allows functions with
  - Same function name
  - Different input arguments (i.e., signatures)
- Example
  - a set of functions that finds the max value among arguments

```
int Max (int a, int b, int c)
{
}

int Max (float a, float b)
{
}

int Max (int a, int b)
{
   if (a>b)
      return a;
   else
      return b;
}
```

- C++ compiler automatically performs Name Mangling
  - Add some postfix to the function name according to the signature
  - E.g, Max_i_i_i(), Max_f_f(), Max_i_i()

Parameter Passing

- Three parameter-passing mechanism
  - Passed by value
  - Passed by pointer
  - Passed by reference

- Passed by value
  - Default mechanism (except for arrays)
  - An object is copied into functions local storage
  - Changing the values of such arguments do not affect the caller

A handy debugging trick

```
#include <iostream>
using namespace std;
#define PRINT(x) cout << #x " = " << x << endl;
void func1(int a, int *b, int &c, const int &d);
int main()
{
   int a=10, b=20, c=30, d=6;
   PRINT(a);  PRINT(b);  PRINT(c);
   func1(a, &b, c, d);
   PRINT(a);  PRINT(b);  PRINT(c);
   return 0;
}
void func1(int a, int *b, int &c, const int &d)
{
   a += d;
   *b += d;
   c += d;
   return;
}
```

By value

```
a = 10
b = 20
c = 30
a = 10
b = 26
c = 36
```

Screen Output

Parameter Passing (cont'd)

- Passed by pointer
  - Efficient when a big object is passed
    - Copying the entire object can be slow
  - Function can change the object being passed
    - Caller clearly know this side effect
- Passed by reference
  - Almost the same as to pass-by-pointer in most cases
    - Efficiency
    - Allow side effects
  - Code is clearer ☺
  - Side effects are implicit ☹

```
#include <iostream>
using namespace std;
#define PRINT(x) cout << #x " = " << x << endl;
void func1(int a, int *b, int &c, const int &d);
int main()
{
   int a=10, b=20, c=30, d=6;
   PRINT(a);  PRINT(b);  PRINT(c);
   func1(a, &b, c, d);
   PRINT(a);  PRINT(b);  PRINT(c);
   return 0;
}
void func1(int a, int *b, int &c, const int &d)
{
   a += d;
   *b += d;
   c += d;
   return;
}
```

By pointer

By reference

```
a = 10
b = 20
c = 30
a = 10
b = 26
c = 36
```

Screen Output

Parameter Passing (cont'd)

- Variant of passed-by-reference — Passed by constant reference
  - Resolve the side effect issues
  - Address are passed
  - Compiler enforces no changes can be made to the objects
- It is typically preferred to use
  - By reference
  - By constant reference
  - Except for passing tiny parameters
    - *char*
    - *bool*

```
#include <iostream>
using namespace std;
#define PRINT(x) cout << #x " = " << x << endl;
void func1(int a, int *b, int &c, const int &d);
int main()
{
   int a=10, b=20, c=30, d=6;
   PRINT(a);  PRINT(b);  PRINT(c);
   func1(a, &b, c, d);
   PRINT(a);  PRINT(b);  PRINT(c);
   return 0;
}
void func1(int a, int *b, int &c, const int &d)
{
   a += d;
   *b += d;
   c += d;
   return;
}
```

By constant reference

```
a = 10
b = 20
c = 30
a = 10
b = 26
c = 36
```

Screen Output