# COMP6771
# Advanced C++ Programming

## Week 1
## Part 2: Introduction to C++

2017

www.cse.unsw.edu.au/˜cs6771

# The Key Concepts in OO

Objects constructed using constructors from types:

- built-in types, e.g., int and float
- user-defined types
  - string, vector, . . . in the library
  - your types!

Sometimes, we want to use a type where another is expected:

- type conversions
- polymorphic types

We perform computations on an object by calling its member functions, written in terms of variables, statements and expressions.

To become a good C++ programmer, we need to have a solid understanding about the basics of C++ . . .

# The C++ Basics

- Types
- Declarations and Definitions
- Scopes and Namespaces
- Expressions and Statements
- Strings
- Arrays
- I/O
- Pointers, References, and `const`
- Lvalues and Rvalues (recurring)
- Functions and Parameter Passing

# What is a Type?

- A set of values, and
- A set of operations

e.g. bool:

- values: {true, false}
- operations: &&, ==, !=, . . .

# Basic C++ Types

1. Primitive types:
    - Built-in types:
        - bool
        - Character types
        - Integer types
        - Floating-point types
        - void
    - Modifiers:
        - Signedness: signed (the default if omitted), unsigned
        - Size: short, long, long long

# Basic C++ Types

2. Constructing new types using declarator operators
   - Pointer types: int*
   - References: double&
   - Array types: char[]
3. User-defined types:
   - The containers such as vector defined in the C++ library
   - All the rest defined by you!

# Booleans

- Two constants: true and false
- non-zero values converted to true
- zero values converted to false

```
bool b1 = 10;     // b1 becomes true
bool b2 = 0.0;    // b2 becomes false

int i1 = true     // i1 becomes 1
int i2 = false    // i2 becomes 0
```

# Character Types

- signed char: type for signed character representation.
- unsigned char: type for unsigned character representation.
- char: type for character representation which can be most efficiently processed on the target system (equivalent to either signed char or unsigned char).
- wchar_t: type for wide character representation
- char16_t: type for UTF-16 character representation (since C++11)
- char32_t: type for UTF-32 character representation (since C++11)

# Integer Types

The C++ Standard guarantees:

```
1 = sizeof(char) <= sizeof(short) <= sizeof(int)
  <= sizeof(long) <= sizeof(long long)
```

The sizes (i.e., ranges) of many types are implementation-defined.

The four widely accepted data models:

- 32 bit systems:
  - LP32 or 2/4/4 (int is 16-bit, long and pointer are 32-bit)
    - Win16 API
  - ILP32 or 4/4/4 (int, long, and pointer are 32-bit)
    - Win32 API
    - Unix and Unix-like systems (Linux, Mac OS X)
- 64 bit systems:
  - LLP64 or 4/4/8 (int and long are 32-bit, pointer is 64-bit)
    - Win64 API
  - LP64 or 4/8/8 (int is 32-bit, long and pointer are 64-bit)
    - Unix and Unix-like systems (Linux, Mac OS X)

# Many Integer Types!

| Type specifier | Equivalent type | Width in bits by data model | | | | |
|---|---|---|---|---|---|---|
| | | C++ standard | LP32 | ILP32 | LLP64 | LP64 |
| `short`<br>`short int`<br>`signed short`<br>`signed short int` | `short int` | at least **16** | **16** | **16** | **16** | **16** |
| `unsigned short`<br>`unsigned short int` | `unsigned short int` | | | | | |
| `int`<br>`signed`<br>`signed int` | `int` | at least **16** | **16** | **32** | **32** | **32** |
| `unsigned`<br>`unsigned int` | `unsigned int` | | | | | |
| `long`<br>`long int`<br>`signed long`<br>`signed long int` | `long int` | at least **32** | **32** | **32** | **32** | **64** |
| `unsigned long`<br>`unsigned long int` | `unsigned long int` | | | | | |
| `long long`<br>`long long int`<br>`signed long long`<br>`signed long long int` | `long long int` (C++11) | at least **64** | **64** | **64** | **64** | **64** |
| `unsigned long long`<br>`unsigned long long int` | `unsigned long long int` (C++11) | | | | | |

# Floating-Point Types

- float: single precision floating point type. Usually IEEE-754 32 bit floating point type
- double: double precision floating point type. Usually IEEE-754 64 bit floating point type
- long double: extended precision floating point type. Does not necessarily map to types mandated by IEEE-754. Usually 80-bit x87 floating point type on x86 and x86-64 architectures

Generally, `double` should be used instead of `float`, due to its higher precision and negligible overhead.

# Range of Values

| Type | Size in bits | Format | Value range: Approximate | Value range: Exact |
|---|---|---|---|---|
| **character** | 8 | signed (one's complement) | | **-127** to **127** |
| | | signed (two's complement) | | **-128** to **127** |
| | | unsigned | | **0** to **255** |
| **integral** | 16 | signed (one's complement) | **± 3.27 · $10^4$** | **-32767** to **32767** |
| | | signed (two's complement) | | **-32768** to **32767** |
| | | unsigned | **0** to **6.55 · $10^4$** | **0** to **65535** |
| | 32 | signed (one's complement) | **± 2.14 · $10^9$** | **-2,147,483,647** to **2,147,483,647** |
| | | signed (two's complement) | | **-2,147,483,648** to **2,147,483,647** |
| | | unsigned | **0** to **4.29 · $10^9$** | **0** to **4,294,967,295** |
| | 64 | signed (one's complement) | **± 9.22 · $10^{18}$** | **-9,223,372,036,854,775,807** to **9,223,372,036,854,775,807** |
| | | signed (two's complement) | | **-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807** |
| | | unsigned | **0** to **1.84 · $10^{19}$** | **0** to **18,446,744,073,709,551,615** |
| **floating point** | 32 | IEEE-754 | **± 3.4 · $10^{\pm 38}$** (~7 digits) | • min subnormal: **± 1.401,298,4 · $10^{-47}$**<br>• min normal: **± 1.175,494,3 · $10^{-38}$**<br>• max: **± 3.402,823,4 · $10^{38}$** |
| | 64 | IEEE-754 | **± 1.7 · $10^{\pm 308}$** (~15 digits) | • min subnormal: **± 4.940,656,458,412 · $10^{-324}$**<br>• min normal: **± 2.225,073,858,507,201,4 · $10^{-308}$**<br>• max: **± 1.797,693,134,862,315,7 · $10^{308}$** |

# Implementation-defined <limits>

```
#include<iostream>
#include<limits>

int main() {
  std::cout << std::numeric_limits<double>::max() << std::endl;
  std::cout << std::numeric_limits<double>::min() << std::endl;
  std::cout << std::numeric_limits<int>::max() << std::endl;
  std::cout << std::numeric_limits<int>::min() << std::endl;
  // ...
}
```

- Performance sometimes comes at the risk of portability
- Must limit the impact of machine-specific language features

# C++ Literal/Constants

- Boolean literals: `true, false`
- Character literals: `'a', '\n', '\r', '\t', L'a'`
  - L'a'stored in wchar_t
- Integer literals: `20, 024, 0x14, 20U, 20L, 20UL`
- Floating-point literals: `12.3, 123E-1F, 1.23E1L`
  - Floating-point literals default to type double

# Advice on Deciding Which Types to Use

For most programs,

- Use an unsigned for non-negative values
- Use int for integer arithmetic
- Use double for floating-point computations
- Use char or bool only to hold characters or truth values.

One more basic type?

**`void`**

A special type with no associated values or operations. It is most often used to specify empty return values for functions.

# Caution: Don't mix signed and unsigned types

```
#include <iostream>

int main() {
  signed int a = -1;
  unsigned int b = 1;
  std::cout << a * b << std::endl;
}
```

What's the output?

# Caution: Don't mix signed and unsigned types

```
#include <iostream>

int main() {
  signed int a = -1;
  unsigned int b = 1;
  std::cout << a * b << std::endl;
}
```

4294967295

-1 is promoted to unsigned int, but the binary representation stays the same: 11111111111111111111111111111111 (32 bits)

The unsigned value is: 4294967295, therefore

$4294967295 * 1 = 4294967295$

Read carefully §2.1.1 – 2.1.2.

# C++: Expression Evaluation Order

- Syntactically similar among C, C++ and Java (but beware of some semantic differences!)
- Should familiarise yourself with the C++ Operator Table
- The order for evaluating the two operands of a binary operator is undefined in the C/C++ language

```
int i = 2;
int j = (i = 3) * i;
std::cout << j << std::endl; // 6 or 9
a[i] + i++; // undefined
foo(a[i], i++); // undefined
f() + g();      // undefined
```

- The exceptions are "&&", "||", the comma operator ","

# C++ Operators

A list of common operators in order of precedence:

- `x.y`, `x->y`, `x[y]`, `x++`, `x--`
- `*x`, `&x`, `++x`, `--x`, `!x`, `~x`, `sizeof`
- `x * y`, `x / y`, `x % y`
- `x + y`, `x - y`
- `x >> y`, `x << y`
- `x < y`, `x <= y`, `x > y`, `x >= t`
- `x == y`, `x != y`
- `x && y` (short-circuit left to right)
- `x || y` (short-circuit left to right)
- `x = y`, `x += y`, `x -= y`
- `x ? y1 : y2` (only one of y1 and y2 evaluated)

# Declarations vs Definitions

C++ programs typically are composed of many files.

**declaration** makes known the type and the name of a variable

**definition** is a declaration, but also allocates storage for, and possibly initialises, a variable

In C++ a variable must be defined exactly once and must be defined or declared before it is used.

**The `extern` keyword**

It declares a variable without defining it (i.e., does not allocate storage), except when it is initialised.

```
extern int i;      // declaration
extern int i = 1; // declaration & definition
int j;             // declaration & definition
```

# Declarations

- A declaration:
  - introduces a name and its type into the program
  - tells the compiler the kind of entity the name refers to
- A variety of declarations are possible:

```
1. extern int error_no;// object declaration
2. typedef int myint;  // typedef declaration
3. int getX(Point *p); // function declaration
   or extern int getX(Point *p)  ;
4. class stack;         // class declaration
5. template <typename T> class Node;
                        // template declaration
```

# Definitions

- A definition defines the kind of entity a name refers to

  1. `int error_no = 20;`
     `entity: the memory allocated for object error_no`
     `define: allocate memory`
  2. `typedef int myint;`
     `entity: int`
     `define: myint is a synonym for int`
  3. `int getX(Point *p) { return p->x; }`
     `entity: the specified function`
     `define: specifies the function body`
  4. `class Book {...}`
     `entity: a data type`
     `define: a list of its data/member functions`

- Every name must be defined before it is used
- The same entity can only be defined once

# C++ Must Distinguish Definitions from Declarations

What if in a file you want to refer to an entity defined in another?

In C++, to allow users to refer to some definitions in a 3rd-party (implementation) library, its developer will provide #include files, i.e., an interface containing their declarations.

**Definitions vs. Declarations in C++**

- Every definition is also a declaration
- There can be multiple declarations for an entity;

# Declarations and Definitions using Third-Party Libraries

- third-party.h

```
extern int count;
extern int doit(int); // extern here is optional
```

- user.cpp

```
#include <iostream>
#include "third-party.h"
int main() {
  std::cout << doit(++count) << std::endl;
}
```

- third-party.cpp (only binaries available)

```
int count = 1;
int doit(int i) {
  i++;
  return i;
}
```

# Header Files

Header files are used to store related declarations and help us structure source code into multiple files. Header files allow us to make full use of C++'s support for separate compilation.

Header files:

- should contain only logically related declarations
- should generally not include definitions
- may include class definitions, some `const` objects and inline functions
- should always be enclosed by header guards

Section §2.9 deals with header files in more detail.

# C++ Preprocessor

The C++ preprocessor inherits the complex features of the C preprocessor, but most of those are not used. We commonly use three features:

The file inclusion directive:

- `#include <standard_header>`
- `#include "my_file.h"`

And the definition and conditional directives:

```
#ifndef MYHEADER_H
#define MYHEADER_H
...
#endif
```

Preprocessor variables are used for avoiding multiple inclusions.

# What Are Found Typically in a Header File

```
class Stack;           // class declaration
class Stack { ... }; // class definition
template <typename T> class stack;
template <typename T> class stack { ... }
inline void f() { ... }
int* g(int);
extern int i;
const double pi = 3.1415926;
enumeration Color { RED, GREEN, BLUE };
typedef void* (*f)(*int) FP;
#include <string>
```

# What Cannot be Placed in a Header File

- No variable definitions – i multiply defined

```
header.h:
      int i = 1;
file1.cpp:
      #include "header.h"
file2.cpp:
      #include "header.h"
```

- No function definitions – f multiply defined

```
header.h:
      void f() { ... }
file1.cpp:
      #include "header.h"
file2.cpp:
      #include "header.h"
```

# One-Definition-Rule (ODR)

- An entity has exactly one definition in a program
- But the entity need not to be defined if not used

```
int f(int); // known as ZDR (Zero Definition Rule)
extern int i;
class Stack;
int main () { Stack *sp; }  // compiles ok
```

- Ideally, a definition should reside in exactly one file, but this is difficult to enforce to due separate compilation

# Some Violations of ODR

```
(1)
file1.cpp:                       file2.cpp
class Point { int x; int y;}  class Point { int a; int b;}
(2)
file1.cpp:                       file2.cpp
class Point { int x; int y;}  class Point { int x; char y;}
(3)
file1.cpp:                       file2.cpp
class Point { int x; int y;}  class Point { int x; }
```

- These programs compile under most implementations – errors not caught by linkers
- Use #include to reduce this kind of violations

# One-Definition-Rule

- A head file can contain:

```
class declarations and definitions
template declarations and definitions
inline function definitions
variable (data) declarations
```

- but not

```
variable (data) and function definitions
```

- Why allowing definitions in header file at all?
  - Inline functions
  - Class definitions
  - Class templates

# Revision: Pointers

- Pointers are variables that hold the address of an object! They allow indirect access to the objects they point to.
- A pointer has a type, a name, a value (an address!) and an address of its own!
- We obtain the address of an object by using &, the address-of operator on that object.
- We use *, the dereference operator to obtain the object a pointer points to (called pointee).

```
int i = 10;
int *p = &i;
std::cout << *p << std::endl;   // *p is an alias of i
                                // i, i.e., 10 printed
```

# Pointers

To minimise bugs, always initialise pointers when they are defined. If the intended address is not available, then set to nullptr.

```
int *p1 = nullptr;
```

**Caution**

Your code will crash if you defernce a nullptr or uninitialised pointer.

The * modifies a variable not the type, be aware:

```
int* p, q; // SAME as int *p, q BUT NOT AS int *p, *q;
```

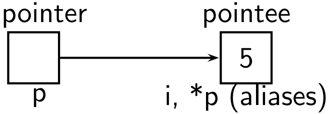
A type mismatch will be detected by the compiler:

```
int *i;
long *lp = &i;  // error
```

Carefully read §2.3.2

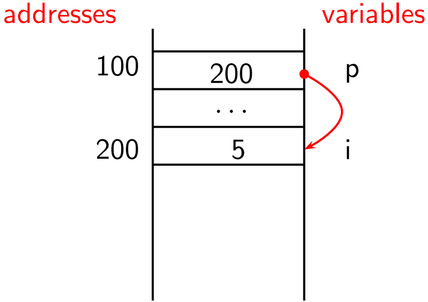
# A Pointer and Its Pointee (or Pointed-to Object)

- Each pointer is associated with two objects:




- Runtime stack:

```
void f() {
  int i = 5;
  ...
  int *p = &i;
  // int *p;
  // p = &i;
}
```



# Pointers Can Be Complex

```
double **p;          // pointer to pointer to double
char* a[10];         // array of 10 pointers to char
char (*a)[10];       // pointer to array of 10 chars
int* (*fp)(int*);    // pointer to function taking
                        a int* and returning a int*
int X::*pm;          // pointer to a member of
                        class X of type int
```

- "`char* a[10]`" is the same as "`char *a[10]`"
- `[]` has the higher precedence than `*`

The misuse of pointers is a major source of bugs!

Revise the pointers if you are rusty

# const

- A variable:

```
int x = 1;
x++;
```

- A const variable:

```
const int y = 1;    // equivalent to C's #DEFINE y 1
          // must be initialised when defined
y++;   // error since it is not modifiable
```

# const

- Initialises a const variable with a non-const:

```
int a = 1;
const int b = a;
```

- Initialises a non-const variable with a const:

```
const int a = 1;
int b = a;
```

# Immutability

- The primary role of const is to specify immutability
- Many objects don't have their values changed after initialisation:
    - Many pointers are often read through but never written through
    - Most function parameters are read but not written to:

```
char* strcpy(char* dest, const char* src);
int strcmp(const char *lhs, const char *rhs );
```

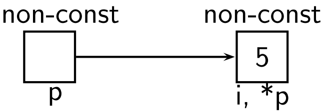
# Pointers and `const`

Four combinations:

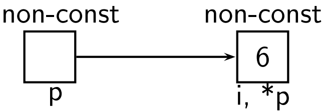
```
Case 1: int *p = &i;
Case 2: const int* p = &i;
Case 3: int *const p = &i;
Case 4: const int *const p = &i;
```

# Case 1: no const

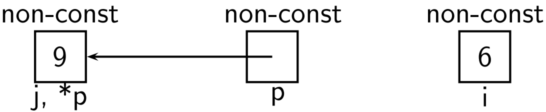
```
int i = 5;
int *p = &i;
```




- After `*p = 6`:

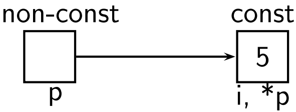



- After `int j = 9; p = &j`:

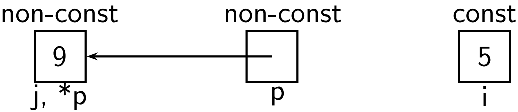


# Case 2: read only dereference

```
int i = 5;
const int *p = &i;
```




- `*p = 6` doesn't compile any more
- After `int j = 9; p = &j`:

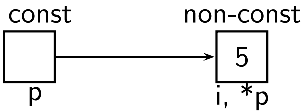



**The promise/contract is to not change the pointee**

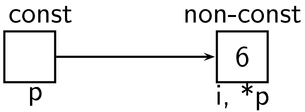
The pointee of p can be either a const or non-const object.

# Case 3: const pointer

```
int i = 5;
int *const p = &i; // must be initialised when defined
```




- After `*p = 6`:

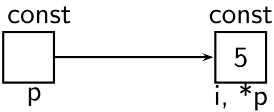



- Any re-assignment for p is prohibited:

```
p = &j; // error as p is a const pointer
```

# Case 4: const pointer & const dereference

```
int i = 5;
const int *const p = &i;
```




- `*p = 6` doesn't compile
- `p = &j` doesn't compile

# const Pointers

A const pointer is *const not const*.

```
char *const cp1     // const pointer to char
char const* cp2     // pointer to const char
const char* cp3     // pointer to const char
```

**Read each declaration right-to left**

cp1 is a const pointer to char
cp2 is a pointer to const char
cp3 is a pointer to "char const", i.e., const char

# References

C++ references are a way of aliasing objects:

```
Case 1: int &r = i;
Case 2: const int &r = i;
```

A reference `r`:

- is just another name for an object called its referent `i`
- must be initialised when defined and may not be rebound to a new object. So a reference is always const.

# Terminology

- A reference is always `const` since it must be initialised when defined and cannot be modified later
- A reference is not itself an object unlike a pointer which is an object.
- A reference to a const object is often said to be a const reference
- A reference to a non-const object is often said to be a non-const reference

**Use references rather than pointers**

They are safer because they can never be null.

# Case 5: non-const or lvalue references

```
// r is an alias of i
int i = 5;
int &r = i;
r = 6;
std::cout << i << std::endl; // 6

const int j = 5;
int &r2 = j;  // error
```

- Must refer to an object of the same type

```
int i;
long &r = i; // error
```

# Case 6: const references

```
// r is an alias of i
int i = 5;
const int &r = i;
r = 6;        // error

const int j = 5;
const int &r2 = j;
r2 = 6; // error
```

- Can refer to an object of a different (but related) type

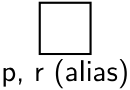
```
int i;
const long &r = i;
const long *p = &i; // error, if i is an int
```

- Can be initialised with a literal

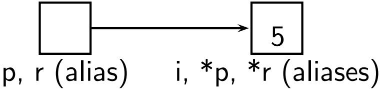
```
const int &r2 = 100;
// REASON: r2 cannot be modified
```

# Mixing Pointers and References

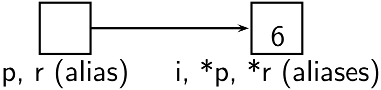
```
int i = 5;
int *p;
int *&r = p;
```




```
r = &i; // SAME AS p = &i;
```




```
*p = 6;
```



# The auto Type Inference

- Let the compiler infer the type for us:

```
auto i = 0, *p = &i;    // ok: i is int and p is int*
auto sz  = 0, pi = 3.14; // error: inconsistent types
```

**The auto Deduction**

Take exactly the type on the right-hand side but strip off the top-level const and &. But the low-level const must be preserved.

# The `auto` Type Inference

- Pointer variables:

```
int i;
const int *const p = i;
auto q = p;                   // const int*
auto const q = p;             // const int* const
```

- References:

```
const int &i = 1; // int
auto j = i;       // int
const auto k = i; // const int
auto &r = i;      // const int& (low-level const kept)
```