C++ 数据类型使用笔记

在 C++ 上机编程和竞赛中，正确选择数据类型非常重要。不同的题目和测试数据可能要求我们使用不同范围和精度的数据类型。下面是一些常见的情境和对应的合适数据类型选择的笔记。

1. 整数类型

1.1 `int`

- **范围**：在 32 位系统上，`int` 的值范围是从 `-2,147,483,648` 到 `2,147,483,647`。
- **使用场景**：通常用于整数范围内的计算，适用于大多数常规问题，尤其是小范围的整数。
- **注意事项**：在处理大数据时，`int` 类型可能会溢出，应当谨慎使用。

```
int a = 100;
```

1.2 `long` 和 `long long`

- `long`：
  - 在 64 位系统上通常是 8 字节，范围约为 `-9,223,372,036,854,775,808` 到 `9,223,372,036,854,775,807`。
  - 在 32 位系统上通常与 `int` 类型相同，4 字节，范围为 `-2,147,483,648` 到 `2,147,483,647`。
- `long long`：
  - 64 位系统中，`long long` 为 8 字节，支持非常大的整数，范围为 `-9,223,372,036,854,775,808` 到 `9,223,372,036,854,775,807`。
  - 是处理大整数问题时常用的数据类型。
- **使用场景**：当数据范围超过 `int` 时，使用 `long long`。例如，处理大数组的下标，计算大的乘积和和等。

```
long long large_num = 1000000000000;
```

1.3 `unsigned` 整数

- `unsigned int`：用于非负整数，范围从 `0` 到 `4,294,967,295`（32 位）。
- `unsigned long long`：适用于需要更大非负整数范围的问题。
- **使用场景**：当确定数据没有负数时（如计数、编号等），使用无符号整数。节省内存空间和提升效率。

```
unsigned int count = 1000;
```

1.4 `short` 和 `long long`

- `short`：通常为 2 字节，范围 `-32,768` 到 `32,767`。适合小范围的整数。
- `long long`：处理非常大的整数，范围比 `int` 和 `long` 更大，适合处理大规模数据。

2. 浮点类型

2.1 `float`

- **范围**：大约 `-3.4e38` 到 `3.4e38`，精度为 6-7 位有效数字。
- **使用场景**：用于存储较小范围的浮点数，节省内存。
- **注意事项**：`float` 精度有限，不适用于需要高精度的场景。

```
float pi = 3.14159;
```

2.2 `double`

- **范围**：大约 `-1.7e308` 到 `1.7e308`，精度为 15-16 位有效数字。
- **使用场景**：常用的浮点数类型，适用于大多数需要浮动范围和较高精度的计算问题。
- **注意事项**：`double` 提供比 `float` 更高的精度，但占用的内存更大（8 字节）。

```
double e = 2.718281828459;
```

2.3 `long double`

- **范围**：通常比 `double` 大，具体范围取决于编译器。大部分系统上为 10 字节。
- **使用场景**：需要极高精度的计算，特别是在科学计算中。

3. 字符类型

3.1 `char`

- **范围**：通常为 1 字节，`char` 类型用于存储单个字符（如 ASCII 字符）。
- **使用场景**：用于存储字符数据，常用于字符串的操作中。
- **注意事项**：`char` 为有符号类型（-128 到 127），如果需要更大范围的字符值，使用 `unsigned char` 或 `wchar_t`。

```
char letter = 'A';
```

3.2 `wchar_t`

- **范围**：用于表示宽字符，通常为 2 字节（在 16 位编码系统下），但也可以是 4 字节（在 32 位编码系统下），支持 Unicode 字符。
- **使用场景**：适合处理国际化字符集，尤其是在需要支持多种语言的应用中。

```
wchar_t symbol = L'字';
```

3.3 `unsigned char`

- **范围**：0 到 255（无符号字符），通常用于处理字节数据。

```
unsigned char byte = 255;
```

4. 布尔类型

4.1 `bool`

- **范围**：只允许 `true` 或 `false`。
- **使用场景**：用于表示逻辑值。

```
bool is_valid = true;
```

5. 其他特殊类型

5.1 `void`

- **用途**：表示没有返回值的函数或指针类型。通常用于函数返回类型或指针类型。
- **使用场景**：函数没有返回值时，使用 `void` 类型；指向任意类型的指针可以使用 `void*`。

```
void printMessage() {
    std::cout << "Hello!" << std::endl;
}
```

5.2 `std::string`

- **用途**：用于存储动态长度的字符串（字符序列）。
- **使用场景**：处理输入输出字符串，尤其是变长字符串数据。

```
std::string name = "Alice";
```

6. 自定义数据类型

6.1 结构体（`struct`）

- **用途**：用于存储不同类型的数据，可以根据问题需求定义合适的结构体类型。
- **使用场景**：当问题涉及多个相关数据元素时，使用结构体来组织数据。

```
struct Point {
    int x;
    int y;
};
```

6.2 枚举（`enum`）

- **用途**：为一组常量提供有意义的名称。
- **使用场景**：当问题中存在离散的状态或标记时，使用枚举类型。

```
enum Color { RED, GREEN, BLUE };
```

7. 指针与引用

7.1 指针

- **用途**：用于存储变量的内存地址。
- **使用场景**：当需要动态分配内存或传递大数据时，使用指针来优化性能。

```
int* ptr = &a;
```

7.2 引用

- **用途**：引用是一个变量的别名，提供了更简便的引用传递方式。
- **使用场景**：当需要直接修改函数参数的值时，使用引用。

```
int x = 5;
int& ref = x;
ref = 10;  // x的值变为10
```

总结

- **整数类型**：`int` 和 `long long` 常用于整数计算，选择时需根据数据范围和精度要求来判断。
- **浮点类型**：`float` 用于小范围，`double` 用于常规计算，`long double` 用于超高精度。
- **字符类型**：`char` 用于普通字符，`wchar_t` 用于宽字符，`unsigned char` 用于处理字节数据。
- **其他类型**：`bool` 用于逻辑判断，`std::string` 用于处理字符串，结构体和枚举用于自定义数据类型。

在上机考试和编程竞赛中，合理选择数据类型能有效避免溢出、精度丢失和内存浪费，提高代码的运行效率和准确性。